ISSN 0265-2919

90p 86

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

A MUSICAL INTERLUDE Increasingly, microcomputers are being used by both the home and professional musician. We take a look at this popular application 1701

HARDWARE

DYNAMIC PERFORMANCE Our guide to computer components continues with a look at how dynamic RAM works 1710

SOFTWARE

COMING OUT OF ITS SHELL The GEM environment is a product of two main streams of development: the graphics kernel system and the SMALLTALK desktop metaphor 1704

WHAT'S ON THE MENU? A look at MicroPro's WordStar — the industry's most successful word processsing package 1718

COMPUTER SCIENCE

COMMERCIAL LANGUAGE The file handling facilities of COBOL make it an ideal choice for commercial applications 1706

JARGON

FROM SYMBOLIC ADDRESSING TO SYSTEMS ANALYSIS A weekly glossary of computing terms 1709

PROGRAMMING PROJECTS

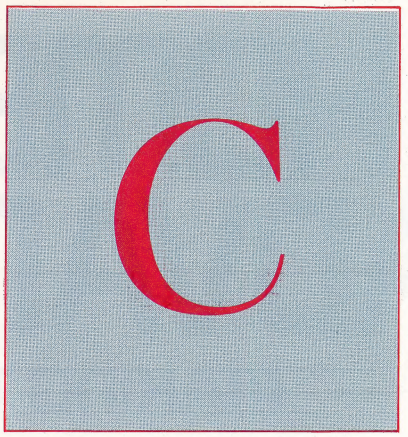
IT'S A DEAL Our latest project is to develop a program that plays a good hand of pontoon 1712

MACHINE CODE

THE CORRECT FORM OF ADDRESS We begin a discussion of the addressing modes available on the 68000 1715

Next Week

- **A language that is finding increasing popularity among programmers is C. We begin a new series investigating the computer language with the shortest name.**
- **Expert systems are gradually gaining ground in the marketplace. We examine Expert-Ease, a package that allows you to create your own expert system.**
- **The 68000 programming series concludes its overview of the available addressing modes.**



QUIZ

1) What is 'step-time programming'?
2) WordStar is installed in a computer as a 'transient program' under the operating system (either CP/M or MS-DOS). What is the main advantage of this arrangement?
3) The 68000 has a 32-bit program counter. How many of these bits are meaningful and why?
4) What is parsing, and what is another name for it?

Answers To Last Week's Quiz

1) The pin groups on an eight-bit processor are the address lines, data lines, system control and CPU control.
2) The extra letters placed at the end of an ADD instruction have the effect of specifying the length (in bits) of the data to be fetched.
3) A 'blitter' is an algorithm that can perform operations on an area of memory independently of the CPU.
4) A word processing language enables the user to write programs which carry out a series of operations, which would normally be performed in direct mode.

Coming Up

- A series on the Unix operating system.
- A look at the use of computers in the entertainment industry.

68000 FAMILY TREE A fact sheet to complement our Machine Code series INSIDE BACK COVER

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistants** Caroline Clayton, Mike Clowes; **Sub Editor** Jon Kaye; **Contributors** Mike Curtis, Steve Cooke, Steve Colwill, Steve Malone, Nigel Cross, David Fensome, Liza Kelly; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 7676, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, Po Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in sterling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER ILLUSTRATION BY ANDY MARTIN

A MUSICAL INTERLUDE

The relationship between mathematics and music theory has long been established, and it's easy to understand the interest in the application of computers to music for both the home and professional musician. We take a broad look here at the development of the links between computers and music.

LIZ HEANEY

Music Boxes
The advent of digitally-controlled sound synthesis has led to an increasing use of home micros as programmable music makers and (at the more serious end of the market) electronic instrument controllers. Software writers have been quick to realise the difficulties most would-be computer musicians face in directly programming microcomputer sound chips and have produced more friendly packages to simplify the task. Here we show some of the music packages currently available

Music packages for microcomputers have been available since the introduction of the first PET and Apple models at the end of the 1970s. These early examples were somewhat crude and limited in their usefulness but the same period also saw the introduction of similarly crude portable electronic music synthesisers designed for home and stage use. Because Western music is structured according to precise mathematical rules, relying on the exact timing of the generation of sounds in combination with silences, many musicians were quick to see the possibilities.

Initially, the main reason for including sound facilities with home micros was to provide suitable noises for games. But some manufacturers, such as Commodore, Acorn and Amstrad, realised there was a demand for music-making capabilities, and included with their machines quite sophisticated chips for generating and shaping sounds. Other manufacturers, such as Sinclair, kept to the 'beep-type' tone generators. This is why there are very few music packages available for the Spectrum, although there are some hardware accessories that add more useful tone-generation circuitry and supply suitable driving software.

The simplest thing for the budding computer musician is to program the micro's sound chip. Most manufacturers who include sophisticated sound chips also include BASIC commands that allow you to select pitches and durations of notes, and shape the sound via envelope commands. A notable exception to this is Commodore, which fails to support its sound chip (arguably the best available on a low-cost home micro) in BASIC — this chip must be programmed using a complicated series of PEEKs and POKEs directly into the sound chip's internal registers.

SOUND CHIP FEATURES

Most sound chips feature three voices, enabling up to three notes to be played at once. Chords and three-part pieces can therefore be easily produced. Volume envelope control affects the quality of a note and is usually defined by a long string of numbers, in turn defining the height and step size of each envelope section. Shaping tone envelopes allows the addition of more sophisticated effects such as vibrato. Furthermore, different waveforms, such as triangle and square wave, are often available.

The main problem with programming music from BASIC is that it is quite slow compared with the degree of acute timing control needed to produce natural sounding music. Even the Amstrad's interrupt-driven sound queue system only alleviates the problem slightly. The best solution (from a programming standpoint) is to use machine code, but of course this makes life extremely difficult for all but the best programmers. As a result, a number of companies who've seen the growth of interest in music

programming have produced friendly interfaces for people who want to play music.

These packages fall into two main types — those incorporating the computer's sound generating facilities, and those using the computer to control external sound-generating equipment. The most obvious example of the latter is a MIDI network of keyboards controlled by a micro (see page 481). We can subdivide the former category further into packages that require additional hardware, such as a keyboard, and those which are purely based on software.

Many packages are available that set up the computer as a playable real-time instrument — that is, keyboard presses are immediately converted into audible and corresponding sounds. Cheaper packages use the computer's keyboard, but expensive add-ons, such as the Echo (see page 1341), provide a more traditional playing medium. Such packages normally include methods of setting the sound chip to alter the sound quality.

MUSIC PROCESSORS?

This is where microcomputers can add extensively to the musician's armoury, since it's possible to write a piece of music (in step-time) using standard notation or a specialised music language in the same way you would use a word processor. This has the added advantage of enabling you to listen to and edit the piece you're composing. Furthermore, it's possible for the computer to analyse and convert a piece played in real-time into the notation system being used. Once you are satisfied, the finished composition can be stored for future reference and printed in standard notation by an ordinary dot matrix printer. In this case, there's no true parallel in the dedicated synthesiser world apart from the use of dedicated composers and sequencers, or the use of a micro for the same purpose via MIDI, such as Yamaha's CX5M (see page 1029).

Most people who are interested in programming will have attempted to program music and sound effects with varying degrees of success. The main appeal of the majority of music packages is twofold: step-time programming of music can allow even the most amateur of users to produce delicate pieces. The proper musician is also attracted to such packages as they allow experimentation via the familiar medium of standard musical notation with composition and form. Systems such as the CX5M, which combine composing facilities with a means of controlling external electronic musical instruments, maximise the potential of this type of system.

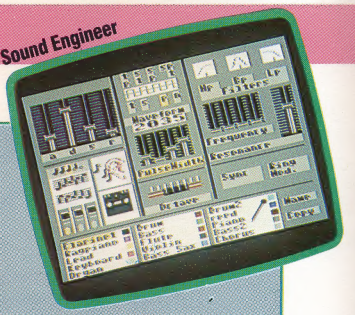
For MIDI instrument users, the relatively small additional expense of a home micro and disk drive provides access to a method of simultaneous control, recording and playback of up to 16 MIDI instruments. Although most MIDI instruments are based on keyboards and drum machines, guitar and wind controllers are also becoming increasingly available.

Music Making Made Easy

Step-time programming is a facility that enables even the complete novice to write his own tunes, which can be played back by the computer. Although there are probably as many versions of step-time programming as there are music composition programs, the essential features are the same.

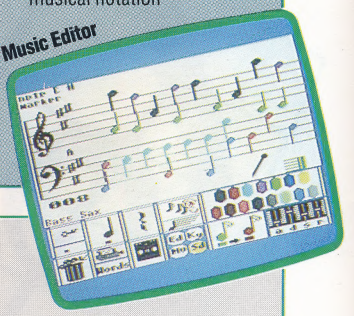
Musical notes are entered into the computer via its own keyboard, and sometimes the note will be played as it is entered. The notes will then be displayed on the screen, often on the standard five-line musical stave. Once the piece has been completed, the user can edit the composition by going back and altering a note, changing the time signature and so on.

The advantage of this method of composition is that users can hear what they are composing as they enter the notes and can have an instant replay. Furthermore, budding composers can hear their tunes, which they may be incapable of playing on a normal musical instrument

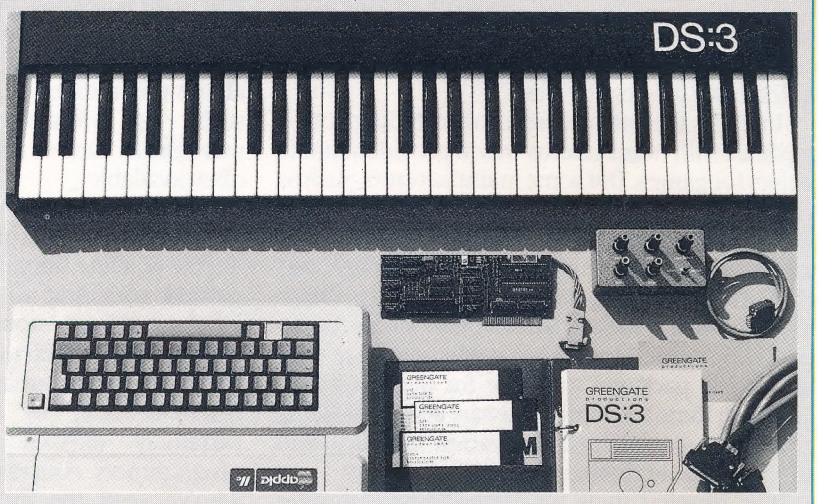


Notable Program

The icon-driven Music Studio from Activision lets you program the sound chip, as well as compose tunes that can be played back. The Sound Engineer has synthesiser-type format, giving you the choice of playing an instrument or programming your own.

The Music Editor program lets you program tunes in step-time. Like the Sound Engineer, features are selected by means of a 'baton' cursor, and the compositions are written in standard musical notation

Sampling

Sampling (see page 581) often comes under the general label of music synthesis, though it's actually the conversion and storage of a sound (an analogue signal) into a series of digital values. These can then be converted back into near enough the original sound on demand, for playing, composing or sequencing. Until recently, such digital 'recording' was limited to very expensive dedicated systems, typically more than £7,000. There's now a system available, the DS3 for the Apple II, that allows four-note polyphonic sampling for £260. Some monophonic systems have been developed for the Commodore 64 (Autographics Microsound 64 at £360) and for the Spectrum (Ricoll Sound Sampler for £200 and Datel DSS for £349.95)



Synthetic Sound

Like computers, the first synthesisers were enormous machines. They were based around oscillator circuits to produce one or two sounds derived from simple pulse, triangle or sawtooth waveforms, singly or in combination. Further circuitry shaped the volume envelope of such sounds and provided different types of signal filters to alter the character of the sounds by eliminating selected frequencies from the signal.

This endowed synthesisers with a degree of versatility but their bulk meant that they could only be installed in studios. Also, their usefulness as musical instruments was somewhat limited because the sounds they produced tended to be 'raspy', thin and colourless. It was on these machines that the idea of synthesisers as keyboard instruments was born. Many different control techniques were tried, from a clarinet-type wind and finger pad system to the interruption of different coloured light beams using cutting motions with the hands. But none were as easy to implement as a piano-like keyboard on which a single note could be signalled 'on' or 'off' by the operation of a simple switch under the appropriate key.

These early portable instruments were monophonic and worked on the same principles as the monster studio models, but saved weight, cost and bulk by using integrated circuits — also, they were still entirely analogue.

Digital techniques were first introduced in the early 1980s in order to stabilise the frequencies generated by the analogue oscillators, since they tended to 'wander' out of tune. This was quickly followed by the addition of memory circuits to store potentiometer and switch settings, enabling the parameters for a particular sound to be recalled and implemented at any time at the touch of a single key. This is essential for rapid changes of sound during a live performance.

Initially, due to the high cost of memory chips, only four to 12 sounds could be remembered. But again, in parallel with the advance of computer technology, modern synthesisers have the capacity to remember hundreds of settings. Also, mass production of sound-generating chips has meant drastic reductions in cost.

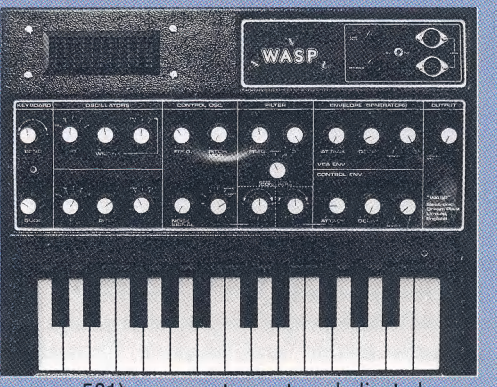
Digitisation of the parameter values for each sound-shaping circuit allowed many manufacturers to cut costs further by eliminating conventional potentiometers, sliders and switches in favour of a minimum of membrane switches. This meant a particular parameter could be 'called' and its current value displayed as an LED number with '+' and '–' keys to change the value. So, by late 1982, almost all synthesisers on the market had their sound parameters digitally definable. As a result, the major manufacturers had to agree on a standard interface and data transfer system so that synthesisers could be connected to each other and computers for control purposes.

A standard system was agreed upon in August 1983 and became known as MIDI (see page 1335). This is really just a software standard whereby MIDI transmissions have certain accepted meanings. The link between two instruments specified by MIDI is merely an asynchronous serial system that had already been implemented successfully for several years on computer systems, allowing them to communicate with peripherals. One of the first models to appear fitted with MIDI was also the first entirely digital portable synthesiser — the Yamaha DX7 (see page 561), a computer system dedicated to the generation and control of musical sounds.

The transmission speeds specified by MIDI are such that a standard home micro can act as receiver, source or intermediate processor of MIDI bytes. The link between computer technology and professional synthesiser equipment has finally been sealed. A number of packages are available that allow you to hook your micro into a MIDI system to let it act as the master control unit in a sophisticated composition, playing and recording system

Mini System
The MiniMoog, pictured here, was the first of a series of inexpensive portable analogue synthesisers based around integrated circuit technology

Cost Cutters
The development of music synthesisers and home computers has several parallels. The Wasp synthesiser pictured here brought the cost of electronic synthesisers within most people's reach by dispensing with the costly mechanical keyboard and replacing it with an inexpensive plastic membrane. The Wasp appeared at around the same time as Sinclair's historic ZX80 and ZX81 home computers, which also featured membrane keyboards

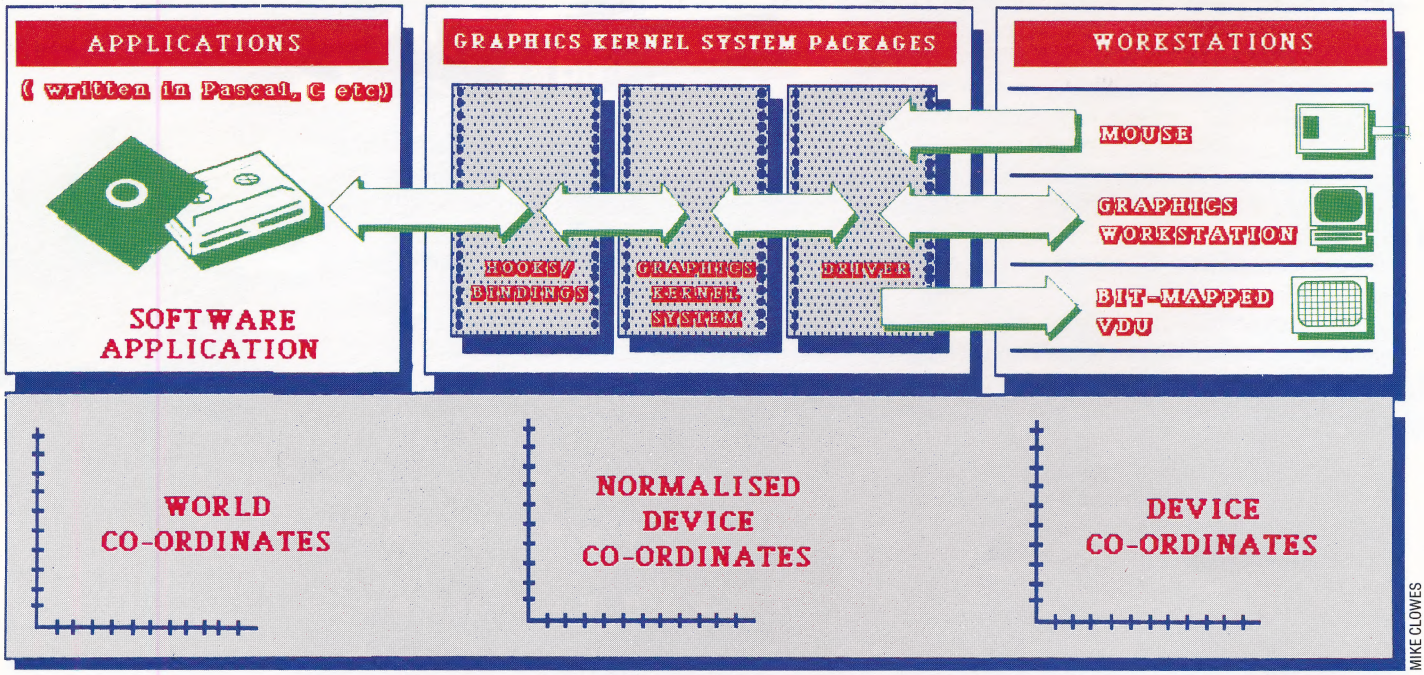


Yamaha DX7

MARCUS WILSON-SMITH COURTESY OF ELECTRONIC MUSICMAKER

COMING OUT OF ITS SHELL

One of the biggest concerns stemming from the implementation of WIMP environments is that of portability. One way around this has been the graphics kernel system specification, which fully illustrates the concept of a portable 'shell'.



MIKE CLOWES

Well Co-Ordinated
GKS and other low-level graphics interface systems achieve portability by translating the real-world co-ordinates input by the user into device co-ordinates acceptable to the hardware. Using this approach, only the device drivers need to be machine-dependent

Implementing a WIMP-like environment for one computer is an expensive operation with a limited market for the final product. Furthermore, it doesn't solve the key problem of making graphics applications fully portable. Portability implies that the normal operating system has a 'shell' around it, controlling the graphics screen windows, pointing device, and also the running of the separate programs selected. When each operation terminates, control is passed back to the supervisory WIMP shell instead of the normal operating system.

In this sense, the controlling program is actually an operating system, but it may still call the native system for the daily house-keeping chores of file management and so on. The diversity of available graphics hardware militates against a consistent environment of this nature, but several schemes for providing graphics shells across a wide range of machines are currently contending for acceptance in the market place, including IBM's TopView and MS-Windows from Microsoft, as well as GEM.

In 1977, the first specifications were published for a 'graphics kernel system' (GKS) that would enable the programming of graphics in a machine-independent way. GKS is now available from several sources, but the complexity and high price restrict its use to serious CAD/CAM applications (see page 1521) and university users. Digital Research was the first company specialising in microcomputers to follow with their GSX (graphics system extension) shell, which was based on GKS, but was not compatible with it.

GSX is an extension to the operating system (DR's own CP/M-86), which is customised for each machine. Once implemented, applications can call GSX routines without worrying or even knowing about the actual hardware, software and firmware facilities used to perform any operation. GSX sits on top of the hardware just as the operating system does, but unfortunately has no user interface — that has to be programmed (with great difficulty!) by the applications writer.

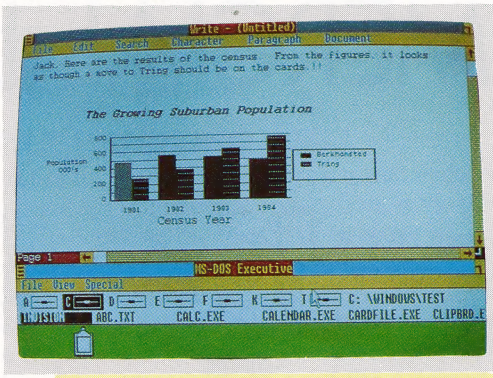
One British software company supplies a library of routines for accessing the GSX facilities without the problems of scaling, co-ordinate mapping and overflow, which are nightmares for the raw GSX programmer. Prospero Software supplies both ISO PASCAL and FORTRAN 77 compilers, and its Prospect library may be linked with programs written in these languages and run unaltered on any machine that has a GSX implementation. This will provide drivers for a wide variety of devices such as mouse, tablet and keyboard inputs as well as printer, plotter and VDU output devices. The Prospect library accesses the GSX facilities including point and line plotting, shading and, if the device allows, text scaling and rotation, line width and colour.

Even with a machine-independent graphics interface and a relatively friendly program library such as Prospero's Prospect, the task of implementing graphics demands a good deal of programming effort. As far as the average user is concerned, the only interaction that can occur is with an application — whether or not it uses

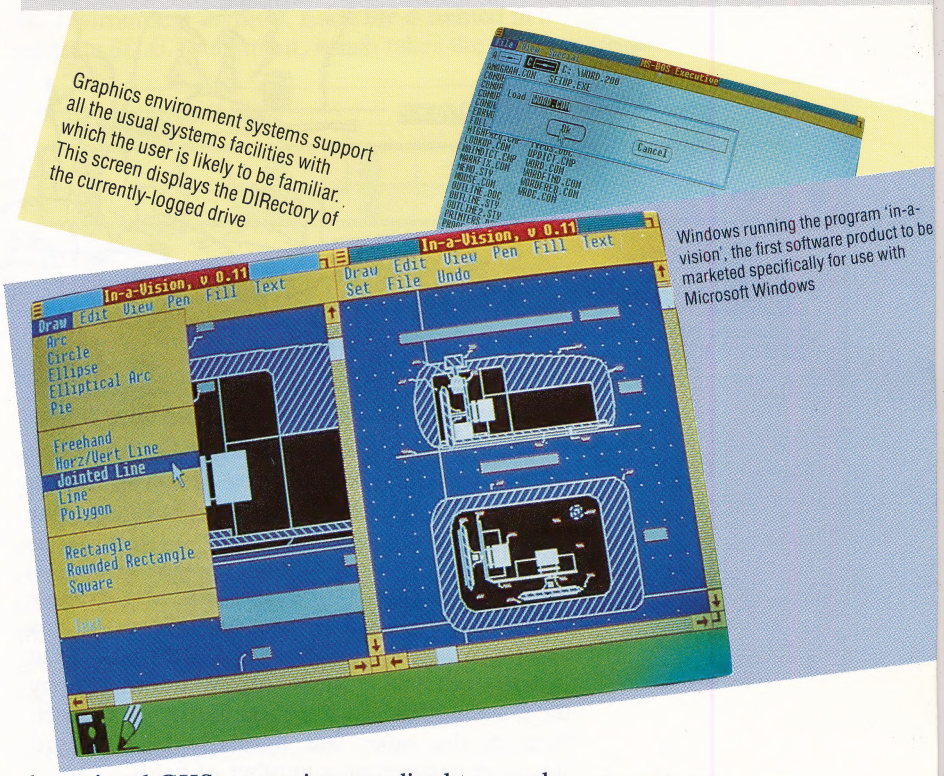
graphics, and whether or not those graphics are driven by GKS, GSX, Prospect, FORTRAN, PASCAL or anything else.

In order to implement a full WIMP system, the whole shell must provide access to every aspect of a computer's resources, not just graphics and not only through individual application or systems programs. Both Digital Research and Microsoft are following this path with GEM and MS-Windows, respectively. Apart from IBM, other hardware manufacturers who are using GEM and MS-Windows on their machines are Apricot, and RML (on the Nimbus).

The graphics kernel system, on which GEM is based, grew out of a desire to develop graphics software that was portable. Digital Research implemented GSX as a non-standard but affordable subset of GKS, and has since taken the idea further with the 'graphics environment manager'. GEM is thus a product of two main streams of development — the GKS standard and the SMALLTALK desktop metaphor popularised by Apple's Lisa and Macintosh.



Colour can make a big difference in manipulating images and high-lighting messages on the screen, even if the application is for monochrome purposes, as here, where a chart developed in one program is transferred to a word processing document



Graphics environment systems support all the usual systems facilities with which the user is likely to be familiar. This screen displays the DIRectory of the currently-logged drive

Windows running the program 'in-a-vision', the first software product to be marketed specifically for use with Microsoft Windows

THE WORKSTATION

Just as CP/M provides a standard operating system through which applications software can control a variable hardware configuration, a graphics kernel such as GEM must wrap a well-defined soft shell around the physical devices in a WIMP system. One of the fundamental concepts is that of the workstation. This can be almost any device from a mouse or plotter to a complete I/O system such as a graphics terminal.

As far as GEM is concerned, each device on the system is accessed exactly the same way and hardware features and constraints are hidden in the 'device driver' for that particular workstation. An output device can therefore be instructed to draw a particular shape whether it is a plotter or a VDU screen. Because the physical construction of each hardware device affects the graphics capabilities and resolution, a system of device-independent co-ordinates is needed to isolate the graphics applications from the hardware. GEM's method is based on GKS which defines three sets of co-ordinates.

Each device must, of course, work to its own set of 'device co-ordinates'. By way of analogy, a colour monitor, for example, will have different cartesian scales on the X and Y axes depending on the VDU's resolution. The actual device co-ordinates and attributes are the only parameters of a system running GEM that vary according to the hardware available.

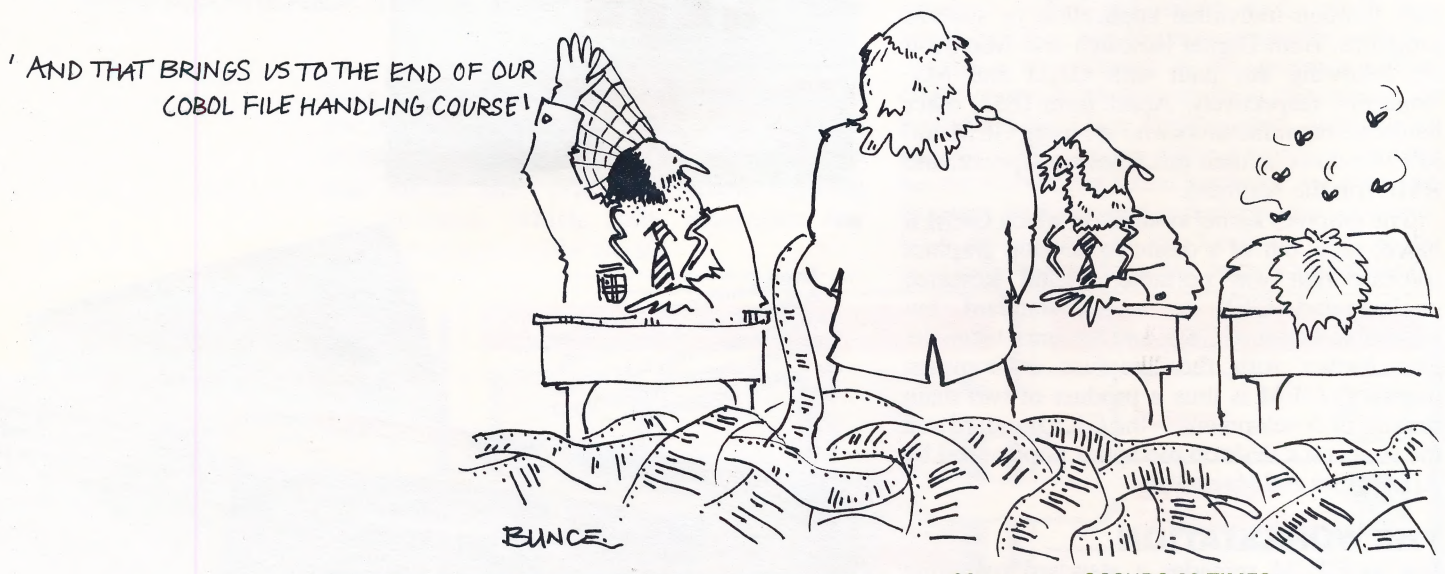
The device co-ordinates are transformed from a set of 'normalised' co-ordinates that are used by GKS for all graphics operations. This second set of normalised device co-ordinates (NDCs) is in turn mapped onto the co-ordinates used by any application software in the real world. These are known as 'world co-ordinates' (WCs) and may be defined by the applications programmer on any scale that is convenient. The NDC level of the three-tiered GKS system is normalised to a scale of real numbers in the range zero to one, so that the only limitation on resolution is real number arithmetic.

On large mainframe computers, the power and speed available alleviate most problems, but things are not as easy with microcomputers. Real arithmetic can be very time-consuming compared to the speed of processing integers (whole numbers), so GEM departs from the GKS system in this respect. The NDC in GEM is scaled from zero to 32,767, allowing each value to be stored and manipulated in one (16-bit) machine word, and enabling the overhead introduced by the transformations to be minimised.

GEM pays a small price for speed, due largely to the integer-normalised co-ordinates used, and is overwhelmingly compensated for by the saving in software development time. Once the initial library of graphics routines is written for GEM, for example, any application may then incorporate them and make full use of the built-in ability to handle the icons, windows and so on. The finished application is then completely portable to any machine in the world that runs GEM.

Window Display
MS-Windows is a graphics environment system produced by Microsoft and tailored especially for compatibility with other Microsoft applications and systems. The screens here show the system in operation. Both GEM and MS-Windows make full use of colour, unlike the Apple Macintosh, but this results in a drop in screen resolution

COMMERCIAL LANGUAGE



We conclude our series on classic languages by considering COBOL's facilities for handling files and tables. We will see just why COBOL has remained such a successful language for commercial applications.

There is a difference between a table and an array which may not at first be obvious — access to items in a table is by means of a KEY value (part of the data in each item in the table) and not directly by means of a subscript or index. Furthermore, each item in the table will normally be a record structure, containing a number of fields.

This sort of structure can be handled in some other languages — PASCAL for instance — by using arrays of records. COBOL has a facility to provide tables, which can be used to provide simple arrays as well. The essential feature is the idea that any data item, except at 01 or 77 level, can be defined as repeating by adding the clause OCCURS nn TIMES, in which nn can be any positive integer constant, and TIMES can be omitted. The following, for example, defines a simple array of 20 integers:

```
01 simple-array.
   02 array-element PIC 9(5) OCCURS 20 TIMES.
```

Elements within the array are referred to in the normal way in statements, as in array-element(5) or array-element(number-1), where number-1 is a positive integer data item. Bear in mind, however, that the entire array also has a name, and so operations such as MOVE can be carried out on the whole array at once.

A two-dimensional array (or larger) can be defined by further splitting up the repeated field into a component that is itself repeated:

```
01 two-dimensional-array.
   02 array-row OCCURS 20 TIMES.
      03 array-element OCCURS 30 TIMES.
```

Elements in this array are referred to as array-element(3,4), for example, or array-element(number-1,number-2); but it's possible to refer to each row, as in array-row(6), or to the entire array.

More complicated structures — ones that make this facility for tables rather than just arrays — can be defined by combining repeated items of various types with the COBOL facility for subdividing data items. For example, the following definition is for a table of prices for various items of stock in a shoe shop that come in 20 different sizes:

```
01 stock-tables.
   02 size-descriptions OCCURS 20 TIMES.
      03 english-size PIC 9V9.
      03 metric-size PIC 99V99.
   02 number-of-items-stocked PIC 999.
   02 items-stocked.
      03 stock-item OCCURS 500 TIMES,
                    ASCENDING KEY IS stock-
                    number.
         04 stock-number PIC X(6).
         04 stock-description PIC X (20).
         04 stock-prices PIC 999V99 OCCURS 20
            TIMES.
         04 stock-indicator PIC X.
         88 in-stock VALUE 'Y'.
```

Note how all the relevant information is kept together in one large table. When a group item such as stock-item in this definition is repeated, all of its subfields are also repeated, so we can refer to stock-number(6) and stock-prices(100,3). Also, the level 88 condition name can be subscripted, so we can use IF in-stock(120). . . . for example. The ASCENDING (or DESCENDING) KEY clause is optional — it's only used when the table is to be kept in

order and then only if the SEARCH ALL verb is to be used. It's the programmer's responsibility to keep the items in order, since COBOL doesn't handle this automatically.

The data items used to index the table can be any numeric item, provided the value it contains is a positive integer. But for convenience and efficiency, COBOL provides a special INDEX data type, and with each OCCURS clause we can add INDEXED BY index-name. This defines a numeric data-item that can only be used to store positive integer values, and to index the array with which it is defined. More than one index can be defined for a table and it's possible to declare any numeric item as USAGE INDEX, in which case it can't be used to index an actual array but can be used in index arithmetic.

Index data items cannot be used in ordinary arithmetic statements but have their own arithmetic verb, SET, which takes the forms:

```
SET index-name TO numeric-value.
SET index-name UP BY numeric-value.
SET index-name DOWN BY numeric-value.
```

where the numeric value can be a constant or any ordinary numeric data item.

SEARCHING A TABLE

One of the main functions that needs to be carried out on a table (as opposed to an array) is searching. The point is that as access is determined by means of a key value, it's necessary to search through the table in order to find the entry with a particular key. COBOL provides this facility directly by means of the SEARCH verb. This has two forms:

```
SEARCH table-name VARYING index-name (or
    numeric-item)
    AT END  imperative-statement
    WHEN condition-1 imperative-statement.
```

where the VARYING clause and the AT END clause are optional with any number of WHEN clauses. This causes a linear search of the table to take place (looking at each item in turn starting from the first), allowing you to specify the action to be taken when a particular condition is met and if the search has finished. An imperative statement is any statement that actually causes a direct action to be taken, with no alternatives. So, for example, an IF is not an imperative statement but a MOVE is.

The alternative form of the SEARCH verb is:

```
SEARCH ALL table-name
    AT END imperative-statement
    WHEN condition-1 imperative-statement.
```

In this case, there can only be one WHEN clause and the condition tested must be of the form:

```
data-item = value
```

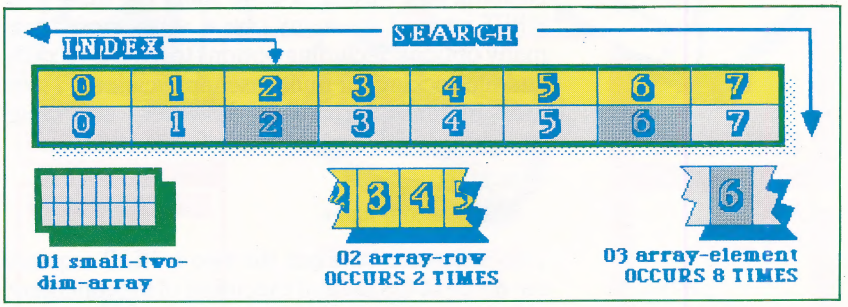
or a number of these connected by ANDs. The difference between the two forms of the verb is that in this case a binary search is made of the table, repeatedly dividing it in half and deciding which half the required item comes in.

In order for this to work, the values in the table must be in order and an ASCENDING or DESCENDING KEY clause must have been specified in the table definition.

FILE HANDLING

One of the main features of COBOL that makes it so suitable for commercial applications is the wealth of file-handling facilities. All versions of COBOL can handle sequential, direct or indexed access to files in a consistent and relatively straightforward way. Files on external devices are initially declared in the input-output section, file-control paragraph of the environment division. Each file is named in a separate SELECT statement, which takes the form:

Indexed Access
COBOL's ability to subdivide data types within the data division enables the user to construct complex records, which may be SEARCHed and INDEXed. Futhermore, the fact that the array itself and each row are declared separately means that block MOVEs can be carried out on data which will encompass the various component items




```
SELECT file-name ASSIGN TO device-name.
```

where the file-name is a COBOL identifier and the device-name is a system-dependent identifier, which may be simply DSK or LPT, or an actual system file name. If the particular version of COBOL does not require the system file name at this point, it will need an extra clause at some point in which this name can be given. If no other information is given at this point, it is assumed that this is a sequential file.

There are a number of optional clauses that allow virtually complete control (where possible) over the way that the file is actually stored. The two most widely used ones define the ORGANISATION and ACCESS. There are three options for ORGANISATION — SEQUENTIAL (the default), RELATIVE (the COBOL name for a file organised for direct access using a record number) and INDEXED. When the organisation of a file is SEQUENTIAL, the only type of ACCESS allowed is SEQUENTIAL. The other two types of organisation, however, also allow RANDOM access (directly to a particular record) or DYNAMIC, which is a combination of RANDOM and SEQUENTIAL.

In the case of INDEXED files, a RECORD KEY must be declared, which will be one of the fields in the data record used by the index. In the case of RELATIVE files, a RELATIVE KEY must be declared, which is a numeric data item used to store the record number.

Each file name mentioned in a SELECT statement must then be defined in the FILE SECTION of the data division, where the name is given in an FD (file definition) declaration along with other system-dependent information such as buffer size, followed by a normal data definition for the record layout.

Although it does seem a bit complicated simply to declare a file, the statements that must be put in the environment and data divisions are standard for the vast majority of applications and have to be learnt just for particular installations.

In the procedure division there are a number of verbs used specifically for file handling. Each file must be opened before use:

```
OPEN file-name FOR access-mode.
```

where the access-mode may be INPUT, OUTPUT or INPUT-OUTPUT; and when the file is finished with it must be closed:

```
CLOSE file-name.
```

Reading and writing is done using the READ and WRITE verbs. Like many COBOL verbs these have many options, including options to handle record- and file-locking for multi-user applications. Their uses in most situations, however, are quite straightforward, and the basic forms are:

```
READ file-name.
WRITE record-name.
```

The difference between the two is READ requires the name of the file and execution of the statement will fill the data record specified for that file in the file section of the data division. The WRITE statement requires the name of that data record and places its contents out on the device specified in the SELECT statement.

If the file has been specified with sequential access, the READ will read the next record from the file and advance to the following record. The end-of-file marker has to be read in to determine whether or not the end has been reached. The READ statement must include the clause AT END, which specifies the action to be taken when this marker is read. The usual way to do this is to use a data item as a flag thus:

```
77 e-o-f PIC X VALUE 'N'.
88 end-of-file VALUE 'Y'.
. . . . . .
PROCEDURE DIVISION.
MAIN-CONTROL-PARAGRAPHS.
    OPEN INPUT in-file, OUTPUT out-file.
    READ in-file AT END MOVE 'Y' TO e-o-f.
    PERFORM process-record-paragraph
        UNTIL end-of-file.
PERFORM close-down.
    STOP RUN.
PROCESS-RECORD-PARAGRAPH.
    . . . . . .
    WRITE out-record.
    READ in-file AT END MOVE 'Y' TO e-o-f.
```

The WRITE will write the new record at the end of the file each time.

When the access to the file is RANDOM or DYNAMIC, and the file organisation is INDEXED or RELATIVE, reading or writing the file is a two stage process. First, an appropriate value must be placed in the key field, and in the case of a RELATIVE file, the record number required is placed in the RELATIVE KEY before issuing the READ instruction.

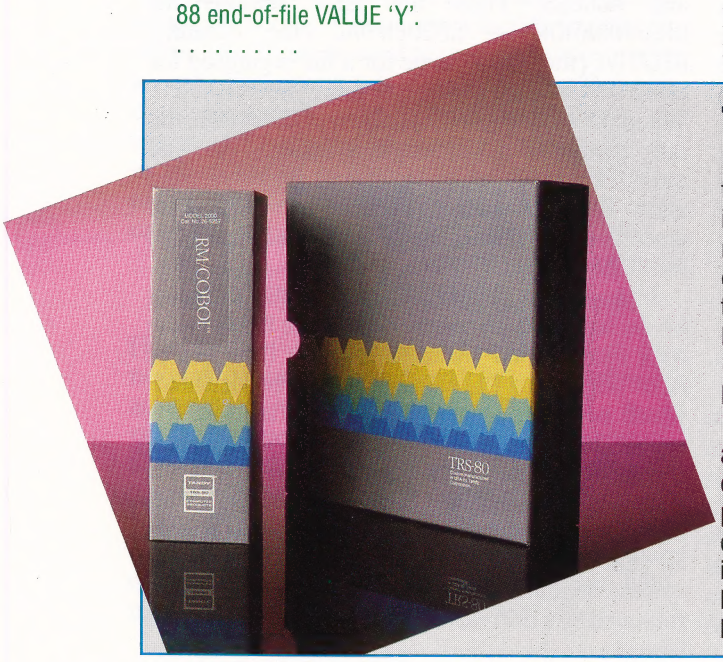
INDEXED files must have an appropriate value placed in the RECORD KEY field. Instead of the AT END clause, there must be an INVALID KEY clause that will be executed if there is no record corresponding to the key value given.

In these two cases, the WRITE verb is only used to write new records. A record that has been updated is written using the REWRITE verb, and a record can be deleted using DELETE. Both these verbs require an INVALID KEY clause.

Sequential access to a file is always possible using:

```
READ file-name NEXT RECORD.
```

This has been a very short introduction to the subject of COBOL file handling which, as one of the strong features of the language, would need a whole series to deal with properly. In fact, most of the language's features have only been looked at briefly here. It's hoped that at least a flavour of the language has been given, providing some general ideas about writing COBOL programs.



LIZ HEANEY SOFTWARE COURTESY OF TANDY

The COBOL Range

Its popularity with the business community means that there are a large number of COBOL implementations for micros, though (because of memory restrictions) very few for home micros. A disk-based system is, of course, essential. Some of the better known versions are CIS-COBOL from Microfocus, Microsoft COBOL and RM/COBOL (shown here). All these run under CP/M and MS-DOS.

Owners of home machines running CP/M — such as the Memotech, Amstrad and Commodore 128 computers — could, of course, use any of these packages (memory allowing), but may find them expensive. As with FORTRAN, a cheaper alternative is Nevada COBOL, available for under £40 from NewStar Software, 45 Plovers Mead, Wyatts Green, Doddinghurst, Essex, CM15 0PS

SYMBOLIC ADDRESSING

The substitution of addresses in an assembly or other source code listing by meaningful names (otherwise known as 'labels') is called *symbolic addressing*. This makes the source code more readable, but more importantly, it enables code to be written so that it is relocatable. Address symbols are actually only replaced with real addresses during the assembly process and are normally either calculated relative to the start address of the program or explicitly defined.

SYNCHRONOUS

A process whereby a series of events takes place at fixed intervals is usually known as a *synchronous* process. These intervals, set by the hardware or under software control, are timed by the computer's or peripheral's clock. A common example of this can be seen in 'synchronous data transfer'. In this case, a computer will transmit packets of data from a register to another computer or device at fixed intervals set by the programmer. The receiving device will be 'synchronised' with the transmitting device so that it is ready to receive the data when it arrives.

SYNTAX

The process whereby the characters and words that make up a programming language are placed in an order understandable to a computer's interpreter or compiler is known as the *syntax*. Failure on the part of the programmer to perform this correctly will result in a 'syntax error'.

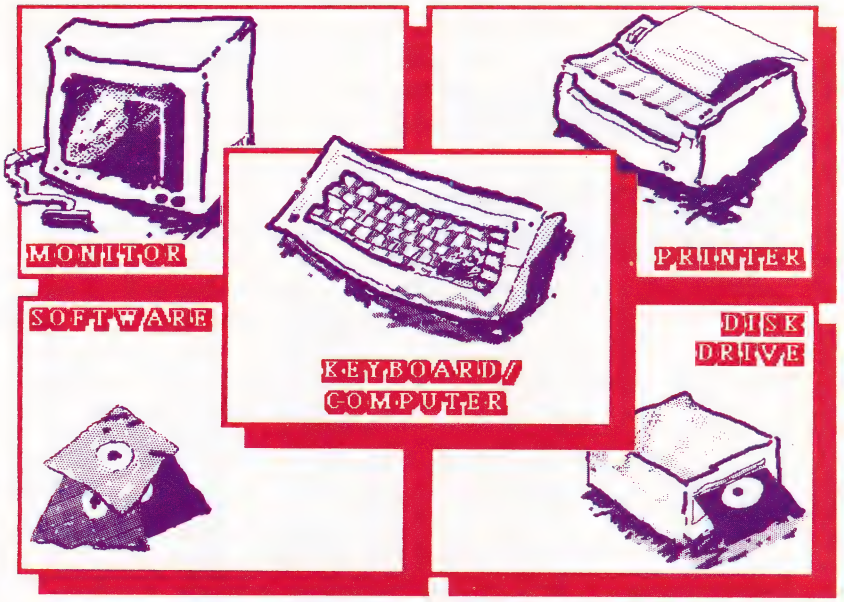
There are essentially two kinds of syntax error, the first of which is simply the misspelling of words. This will occur because the computer has a set number of keywords held in memory. When interpreting or compiling a program, the computer will look at a line of code and try to match the characters on the line with the permitted words in memory. If it fails to find a match, an error will be generated.

The second type of syntax error is slightly more complex. Known as syntax analysis, or 'parsing', it is the process whereby the interpreter or compiler checks to see whether the words that make up the line of a program are in their correct and logical sequence. In order to achieve this, the computer will also hold a number of syntactical 'rules' within the memory. Thus, when interpreting a line, the computer will have certain expectations as to what is coming next. When interpreting a BASIC program, for example, should the computer find the keyword IF on a line, it will expect to come across the command THEN elsewhere on the line, since both form part of the same conditional statement. If one of the words is not present, a syntax error will be generated and the program interpretation will stop.

SYSTEM

A nebulous term, *system* usually refers to the complete configuration of a computer, its peripherals, backing store and the operating system and compiler programs (known collectively as the 'systems software') used to run it. Often, however, the term is used to mean any grouping of hardware and software running together.

From this term, a number of others have been coined. 'System design' concerns itself with the planning of a system from the requirements specified by a user. Obviously, most systems will require the components just listed. But in order to tailor the system to the user's needs, certain components such as networking facilities or additional maths processing may be required while others, like additional back-up memory or a 'front-end' system, may not.



MIKE CLOWES

Parts Of The System
In simplified terms, a computer system is all the hardware and software components used in a specific configuration. Therefore, when we refer to a 'system' we also mean the peripherals that are needed, as well as the necessary programs. Almost by definition this implies that there is an enormous variety of different systems, each one customised to the users' specific needs

'System generation' refers to tailoring an operating system to a user's own requirements. Some operating systems, for example, may require a disk operating system included, whereas in others it is superfluous.

SYSTEMS ANALYSIS

When designing a system, we first have to identify what features are required and the optimum way of organising them. This is known as *systems analysis*. Systems analysts will receive a series of specifications from the user and then decide how these needs can best be implemented on a computer system. This means, for example, that although many database programs are available on the market, the analyst may need to design a database himself in order to achieve maximum efficiency in a given situation. A systems analyst is also required to design the system in such a way that structured programs can be easily constructed.

Once the systems analyst has finished the design, the plans can be passed to a programmer who will then write the code to implement the system. When the program has been written, it will often be passed back to the systems analyst for testing to ensure that the program meets the requirements of the design.

DYNAMIC PERFORMANCE

In the previous instalment of our guide to the components of a computer system, we saw how the processor controls ROM and static RAM. We look now at dynamic RAM, the more common form of memory found in home micros, and delve into the internal workings of a RAM chip.

In the last instalment (see page 1690) we briefly stated that there are two main types of RAM — static and dynamic. While static RAM relies on a small logic device known as a flip-flop, dynamic RAM holds its bits of data as electronic charges.

Both forms of RAM have advantages and disadvantages associated with them. The transistor that is used to hold a single bit charge within a dynamic RAM is much smaller than the flip-flop used to hold the same unit of data in a static RAM. Dynamic RAMs can therefore be packed more densely onto the chip's surface. The charges holding the data in a dynamic RAM, however, will leak away after a few milliseconds and so additional circuitry is required to periodically 'refresh' them. This problem does not exist with regard to static RAM, and so, if the system only requires a small amount of memory, static RAM is the more economic option. Where larger memories are required, the added expense of the refresh circuitry becomes worthwhile and dynamic RAM is more likely to be used.

We saw previously that static RAM is normally arranged as eight-bit registers, with each chip possessing eight data pins linked to the eight data bus lines. Dynamic RAMs tend to be constructed on different lines — typically each dynamic RAM chip will represent one of the eight data bits that make up a location in memory, and eight such chips, wired in parallel, constitute bytes of memory.

The first diagram shows the 4116, with the typical pin connections of a 16 Kbit chip. Many eight-bit micros now use 4164 (64 Kbit) dynamic RAMs of this type to achieve a 64 Kbyte RAM memory. Although we can imagine the chip as being one bit 'wide' and 16 by 1,024 bits 'long', it's actually arranged in 128 rows by 128 columns.

If we contrast the pin-outs from this chip with those from the static ROM (see page 1690), we can see several notable differences. First, rather than eight data pins there are only two — called *data in* and *data out*. We would expect a 16 Kbit chip to require 14 address bits to select any bit uniquely, but in fact there are only seven. Two other unfamiliar lines are also present: $\overline{RAS}$ and $\overline{CAS}$, which are the *row address strobe* and *column address strobe*, respectively. These two timing signals allow the address to be presented in two halves (hence the seven address pins rather than the expected 14). The $\overline{RAS}$ line also serves to refresh the dynamic RAM.

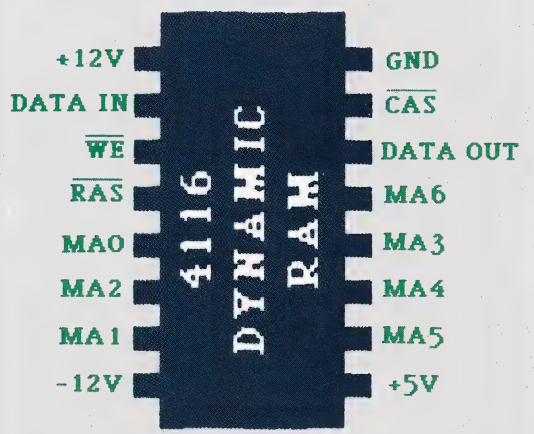
The refresh process consists of reading the data out of the dynamic RAM and writing it back again to restore the charge. In our example RAM, this can be done a row at a time. Thus only 128 operations are needed to refresh the complete chip. Although the refresh will in most cases slow the processor down by delaying memory accesses during the refresh cycles, the retardation is only likely to be less than five per cent.

ADDRESSING A BIT

We mentioned previously that the 14 address bits needed for a 16 Kbyte memory can be presented to our example dynamic RAM in two seven-bit parts. This is achieved through an external logic chip that gates the low and higher order address lines with the CAS timing signal. The second diagram shows a simplified arrangement whereby the low order address lines (A0 to A6) are connected to the chip's address pins when CAS is high and the high order address lines are connected when CAS is low. Thus on the RAM chip itself, there are two latches that have the ability to 'freeze' the data coming into them, and therefore the row address is taken while $\overline{CAS}$ is high and the column address is taken when $\overline{CAS}$ is low. Putting these latched values through 7-to-128 decoders allows the addressed bit to be accessed.

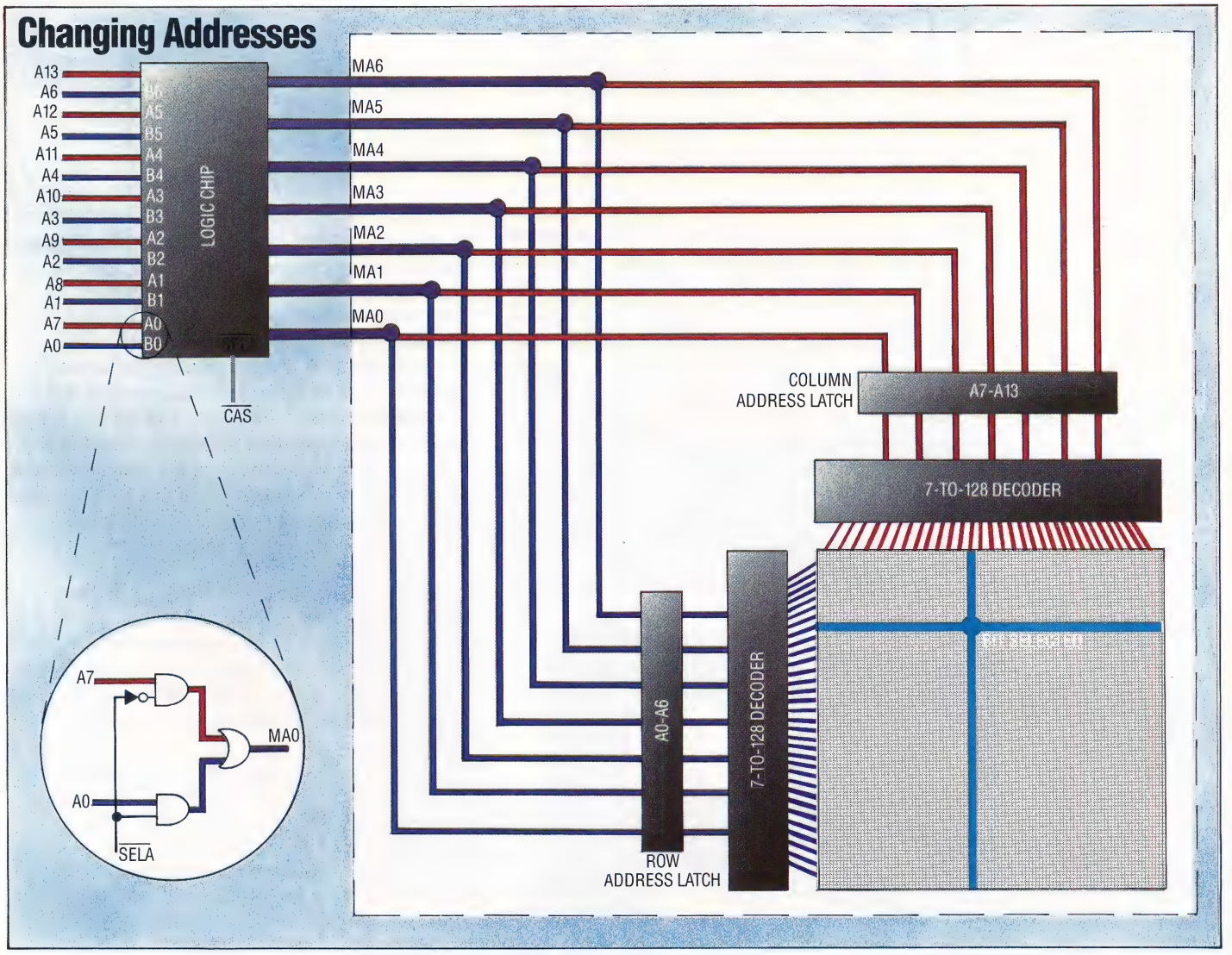
The final diagram shows how a 16 Kbyte dynamic RAM memory is linked to the processor's address, data and read/write lines. The $\overline{CAS}$ line is linked in parallel to each of the eight RAMs and to the addressing logic chip. $\overline{RAS}$ and $\overline{WE}$ are also wired to all eight RAMs. The data in and out pins on each RAM are wired together and connect to one of the eight data bus lines.

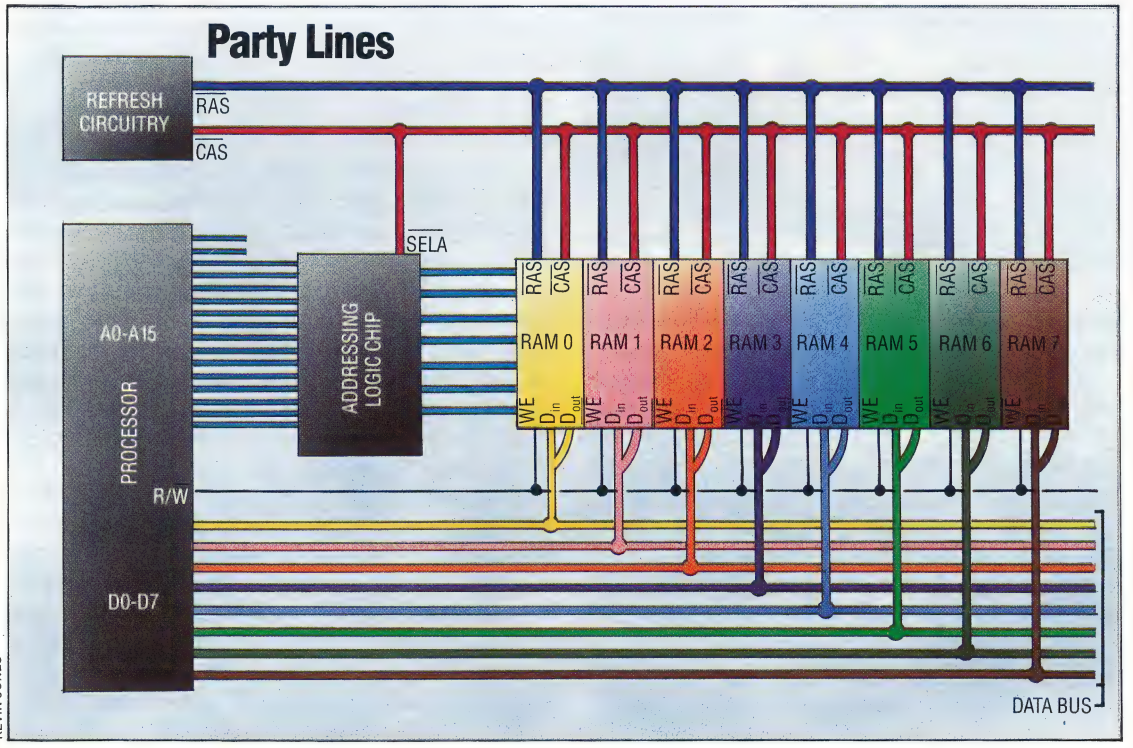
Dynamic RAM



This 16-Kbit RAM has typical pin connections. Unlike a static RAM, a dynamic RAM normally corresponds to a single bit of the data word — hence the single data in and data out lines. Only half the expected number of address lines are present simplifying the wiring of such RAMs considerably

MIKE CLOWES





KEVIN JONES

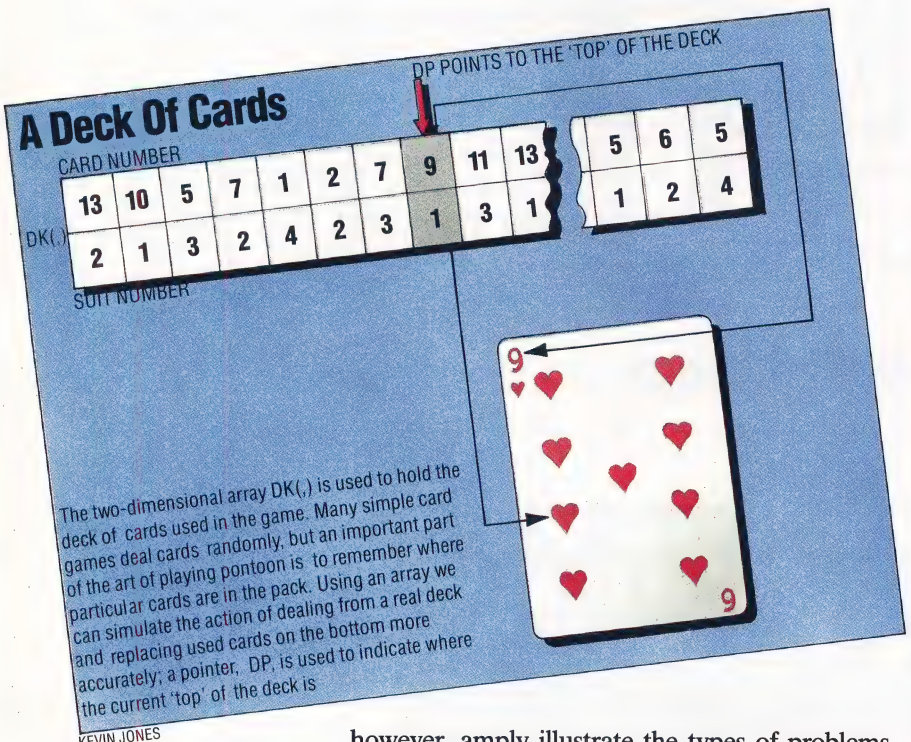
Shared Address
The 14 address lines needed to select each bit in a 16-Kbit RAM are used to produce seven-bit column and row addresses. To simplify circuitry, the chip itself has only seven address pins and the switching of the low and high order parts of the input address are normally handled by an external logic chip (using CAS as a synchronising signal). The inset shows the simple arrangement of combinational logic needed to switch the two address halves

Figures Of Eight
A 16-Kbyte memory can be constructed by wiring eight 4116 chips in parallel so that they share the address, R/W, RAS and CAS lines. In such an arrangement each dynamic RAM is connected to a single bit on the data bus

IT'S A DEAL

We begin a new project to create a program to play the well-known card game of pontoon. First of all, we will design the routines that display the cards on the screen and devise the part of the program that simulates the shuffling of a deck of cards.

Card games are ideally suited for implementation on a computer. Their rigidly defined rules of play and strategies often rely heavily on mathematical analysis. Pontoon, for instance, is particularly easy to program and the rules are very straightforward, making it a simple matter to code for the computer. The principles of creating a deck of cards, displaying them and evaluating hands, however, amply illustrate the types of problems that can be encountered.



The two-dimensional array DK(,) is used to hold the deck of cards used in the game. Many simple card games deal cards randomly, but an important part of the art of playing pontoon is to remember where particular cards are in the pack. Using an array we can simulate the action of dealing from a real deck and replacing used cards on the bottom more accurately; a pointer, DP, is used to indicate where the current 'top' of the deck is

KEVIN JONES

To begin with, let's look at setting up the deck and displaying individual cards, incorporating separate listings for the Commodore 64, Sinclair Spectrum, BBC Micro and Amstrad CPC range. For the time being though, we'll be providing the Commodore version only — the other three will be given in the next instalment.

The simplest method of representing a deck of cards is to hold it as a two-dimensional array. In our game, the array DK(,) is DIMensioned to be 52 elements long and two elements wide. Since there are 52 cards in a normal pack (excluding the jokers) it is obvious why we need 52 elements in one direction. If we think about an individual card it has two properties that make it unique within the pack: its value (ace, 2, 3 and so on) and its suit (hearts, diamonds, clubs or spades). Each card's value and suit are therefore held separately within the array, using the second dimension.

So that the array can be numeric, we've used the numbers 1 to 13 to hold the card value, such that 1 is the ace and 13 the king, and a number from 1 to 4 to represent the suit. The deck array and other arrays used to represent the cards and their positions must be initialised at the beginning of the program, which is done by the subroutine at line 500.

The method we've used to simulate pack shuffling uses a pair of nested loops. The outer one counts through the four suits and the inner one counts through the 13 cards in each suit. Thus the two loop counter variables represent the card suit and card value to be entered onto the deck array within the inner loop. All we need to do is select the point at which we wish to place the current card within the deck.

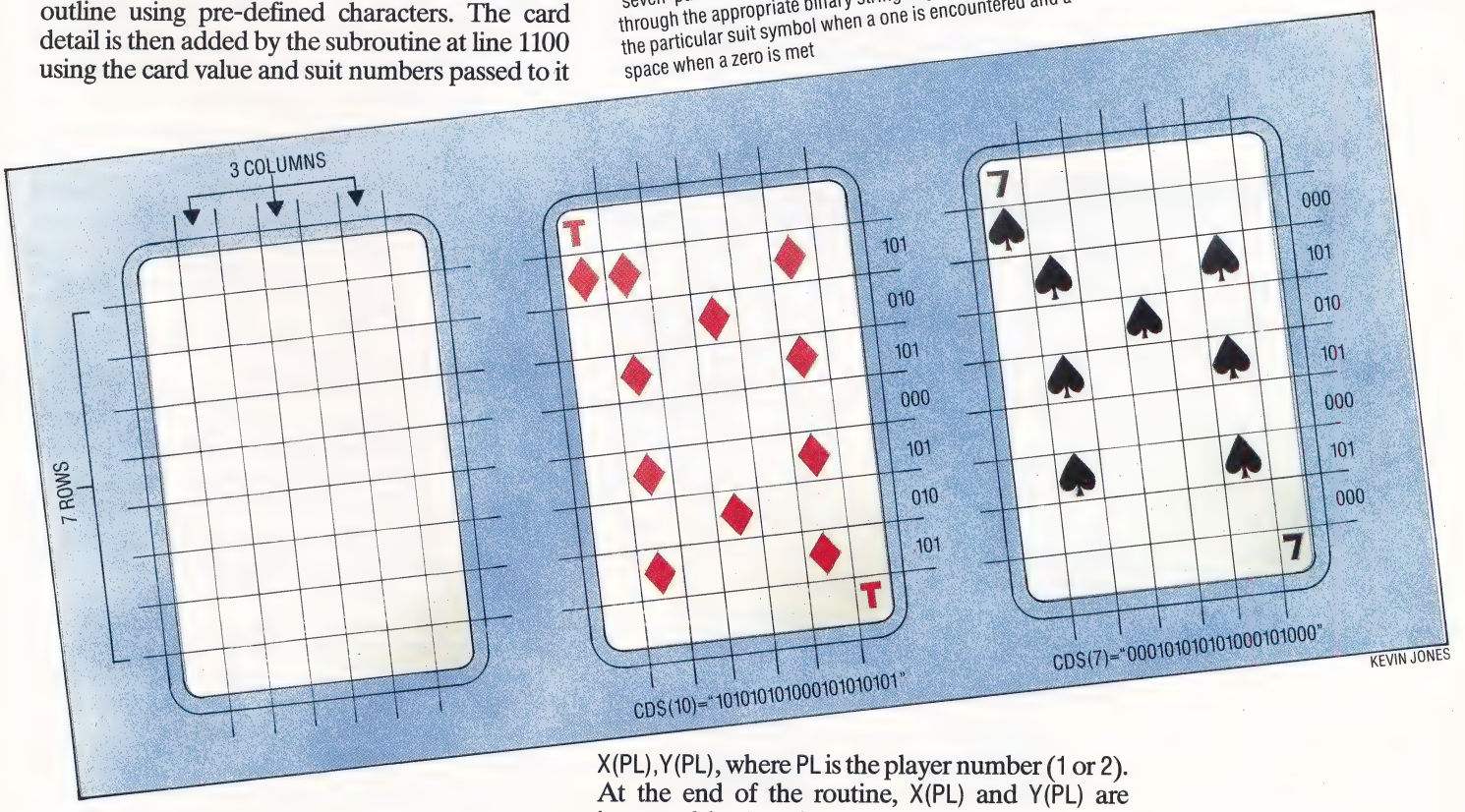
Since the pack is supposed to be shuffled, our routine selects the entry point to the pack randomly, but there's a slight snag here — the position selected in the deck array might already be occupied. To get around this, a small routine within the inner loop tests the randomly selected entry point — if it's not empty, increment the entry point (wrapping back around to the front of the deck if necessary) until an empty space is found. On exit from the nested loop structure, our deck will contain 52 unique cards in a random order.

DISPLAYING CARDS

To display a card from the deck, all we need are the card value and suit. However, we have to interpret these two data items to produce the card pattern and the card corner labels. Although for most cards the corner label will correspond directly to the card value, the ace, jack, queen and king will need A, J, Q and K labels. We also have to cheat slightly by representing the 10 as T, so that the label can be displayed in a single column on the card. The easiest way to handle all this is to set up a card number string array, CN$(), to hold the 13 card labels.

The card pattern is a little more difficult. For simplicity, we'll represent all picture cards as though they were an ace — that is, display a single suit symbol in the centre of the card. If you look at a pack of cards, you'll see that all the patterns are regular, geometric shapes and it is, in fact, quite easy to design a template into which all the patterns can be fitted.

There are three columns and seven rows that



Pick A Card

The patterns for all thirteen card types are stored as binary strings. Using a three column by seven row template, our diagram illustrates how the binary strings for a 'ten' and a 'seven' pattern were created. The card display routine works through the appropriate binary string in groups of three, printing the particular suit symbol when a one is encountered and a space when a zero is met

can possibly hold suit symbols, giving us 21 symbol positions in all, and there are several methods of holding patterns. We've chosen to hold each card pattern as a 21-element binary string, using 1 to represent a displayed symbol and 0 when no suit symbol is present. The definitions for the 13 cards are held in data statements at line 2100 and are read into the array CD$(). We can now look at the card display routine itself.

In the Commodore 64 version given here, there's no method of directly positioning the cursor at a given point on the screen. The program therefore uses a subroutine at line 900 to position the cursor at a point defined by TX and TY.

The subroutine at line 1050 prints the card outline using pre-defined characters. The card detail is then added by the subroutine at line 1100 using the card value and suit numbers passed to it as CN and SU. The first task is to select the suit symbol from a string of the four symbols defined in the initialisation routine. The card colour can be easily found by testing the suit number. Because the suits are arranged in the order hearts, diamonds, clubs and spades, we simply need to set the card colour to black if the suit number is bigger than two, and set it to red otherwise.

The pattern can now be printed from the binary description held in CD(CN). A pair of nested loops are used to step through each of the seven rows, taking the binary string in groups of three and assembling a string of spaces and suit symbols that constitute the current row being worked on. Having printed the card pattern, we can add the corner labels by positioning the cursor in the corner and printing CN$(CN).

At this stage, let's look at how the cards are positioned. During the game, cards will be dealt to the player or the computer and placed in such a way that they overlap but are offset by two units horizontally and one unit vertically. To keep track of the position at which the next card is to be printed, the arrays X() and Y() are used. The first element holds the next card position for the player's hand and the second for the position of the computer's next card. Thus at the beginning of the card printing routine, the cursor is moved to X(PL),Y(PL), where PL is the player number (1 or 2). At the end of the routine, X(PL) and Y(PL) are increased by 2 and 1, respectively, ready for the next card to be dealt to that player.

In our programmed version of the game, the computer plays the part of the bank, so it has its first card dealt face down. We therefore need a little routine to display the back of this card. The subroutine at line 1200 handles this by first calling the card outline routine and then filling in the interior using a chequerboard character.

DEALING A CARD

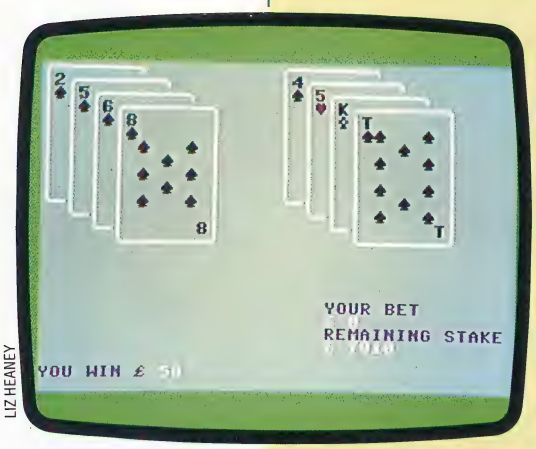
The final routine in this section allows us to deal cards from the top of the pack and display them. The short subroutine at line 1300 is responsible for this. It takes the elements from the deck array that correspond to the top of the pack and puts them into CN and SU, ready for the call to the card display routine.

The obvious way to handle dealing would be to take the first elements from the deck array, move all other elements forward one place and put the

elements just removed in DK(52,1) and DK(52,2). In fact, all this movement of array elements is time consuming and ultimately unnecessary. Instead we can use a variable, DP, that points to the 'top' of the pack and is incremented each time a card is dealt, wrapping round from position 52 to 1 if necessary. To deal a card, we therefore take element DK(DP,1) as the card number and DK(DP,2) as the suit number.

In order to see the effect of our labours up to now, the following lines can be added to call the various routines so that five cards are dealt to both the player and computer:

```
60   PL=1:FOR C=1 TO 10
70   PL=3-PL:REM TOGGLE PLAYER NUMBER
80   FL=0:GOSUB 1300:REM DEAL CARD
90   INPUT'PRESS RETURN FOR NEXT
     CARD';AN$
100  NEXT C
```



LIZ HEANEY

You Need Hands . . .
Our pontoon program displays the cards in the player's and banker's hands on the left and right halves of the screen. Offsetting each newly dealt card sideways and downwards allows all the card values in each hand to be seen. At this stage of the project you will only be able to display the cards. The prompt, message and betting displays are the subject of future instalments

Card Display And Shuffling

Main Program:

```
10 REM **** COMMODORE 64 PONTOON ****
20 GOSUB 500:REM INIT ARRAYS ETC
50 REM **** GAME LOOP ****
55 GOSUB 600:REM INIT GAME
```

Array Initialisation:

```
500 REM ***** INIT ARRAYS ETC ****
510 SP$="":DW$="":FORI=1TO25:DW$=DW$+CHR
$(17):SP$=SP$+" ":NEXT I
511 SP$=SP$+LEFT$(SP$,14)
512 BK$="":LI$="":FORI=1 TO 7:LI$=LI$+CH
R$(195):BK$=BK$+CHR$(166):NEXT I
513 BR$=CHR$(194)
515 DIM X(2),Y(2):REM NEXT CARD POSITION
S
520 SU$=CHR$(211)+CHR$(218)+CHR$(216)+CH
R$(193):REM SUIT TYPES
530 DIM CN$(13):FORI=1TO13:READ CN$(I):N
EXT:REM READ NUMBER DATA
540 DIM CD$(13):FORI=1 TO 13:READ CD$(I)
:NEXT:REM READ PATTERN DATA
560 DIM DK(52,2):REM SET UP CARD DECK AR
RAY
570 GOSUB3000:REM SHUFFLE PACK
580 POKE53280,5:POKE53281,15:REM SCREEN
COLOURS
590 RETURN
600 REM **** INIT GAME ****
605 PRINT CHR$(147):REM CLEAR SCREEN
620 X(1)=0:Y(1)=0:X(2)=20:Y(2)=0
630 RETURN
```

Card Display Routine:

```
900 REM **** PRINT AT ****
910 PRINTCHR$(19);:PRINTLEFT$(DW$,TY);TA
B(TX);:RETURN
1000 REM **** DISPLAY CARD ****
1010 GOSUB 1050:GOSUB 1100:RETURN
1050 REM **** CARD BLANK ****
1055 TX=X(PL):TY=Y(PL):GOSUB 900:REM POS
ITION
1060 PRINT TAB(TX);CHR$(5);CHR$(213);LI$
;CHR$(201)
1070 FORI=1 TO 9:PRINT TAB(TX);BR$;"
   ";BR$:NEXT I
1080 PRINT TAB(TX);CHR$(202);LI$;CHR$(20
3)
1090 RETURN
1100 REM **** CARD DETAIL ****
1120 TX=X(PL)+2:TY=Y(PL)+2:GOSUB 900:REM
 POSITION
1125 CT$=MID$(SU$,SU,1):REM SELECT SUIT
TYPE
1127 CO$=CHR$(28):IF SU>2 THEN CO$=CHR$(
144):REM SELECT COLOUR
1128 PRINTCO$;
1130 FOR I=1 TO 19 STEP 3
1140 CC$=MID$(CD$(CN),I,3):CL$=""
1142 FOR J=1 TO 3:C$=CHR$(29)+CHR$(29)
1144 IF MID$(CC$,J,1)="1" THEN C$=CT$+CH
R$(29)
1146 CL$=CL$+C$:NEXT J
1150 PRINT TAB(X(PL)+2);CL$:NEXT I
1160 REM **** ADD CORNER LABELS ****
1170 TX=X(PL)+1:TY=Y(PL)+1:GOSUB900:PRIN
TCN$(CN):REM NUMBER
1180 TY=TY+1:GOSUB900:PRINTCT$:REM SUIT
1190 TX=X(PL)+7:TY=Y(PL)+9:GOSUB900:PRIN
TCN$(CN):REM BOTTOM NUMBER
1192 X(PL)=X(PL)+2:Y(PL)=Y(PL)+1
1195 RETURN
1200 REM **** DISPLAY CARD BACK ****
1210 GOSUB 1050:GOSUB 1250:RETURN
1250 REM **** CARD BACK ****
1255 TX=X(PL):TY=Y(PL)+1:GOSUB 900:REM P
OSITION
1260 FORI=1 TO 9:PRINT TAB(TX);BR$;CHR$(
156);BK$;CHR$(5);BR$:NEXT I
1270 X(PL)=X(PL)+2:Y(PL)=Y(PL)+1
1280 RETURN
1300 REM **** DEAL A CARD ****
1310 CN=DK(DP,1):SU=DK(DP,2)
1320 DP=D+DP+1:IFDP>52 THEN DP=1:REM BUM
P DECK
1330 IF FL=1 THEN GOSUB 1200:RETURN:REM
DISPLAY BACK OF CARD
1335 GOSUB 1000:RETURN:REM DISPLAY CARD
2000 REM **** CARD NUMBER DATA ****
2010 DATA A,2,3,4,5,6,7,8,9,T,J,Q,K
2100 REM **** CARD DISPLAY DATA ****
2110 DATA"00000000001000000000":REM A
2120 DATA"00001000000000010000":REM 2
2130 DATA"00001000001000001000":REM 3
2140 DATA"00010100000000101000":REM 4
2150 DATA"00010100001000101000":REM 5
2160 DATA"00010100010100101000":REM 6
2170 DATA"00010101010100101000":REM 7
2180 DATA"00010101010101101000":REM 8
2190 DATA"10100010101010100101":REM 9
2200 DATA"10101010100010101010":REM 10
2210 DATA"00000000001000000000":REM J
2220 DATA"00000000001000000000":REM Q
2230 DATA"00000000001000000000":REM K
```

Shuffling The Deck:

```
3000 REM **** SHUFFLE THE DECK ****
3005 R=RND(-TI):DP=1
3007 FORI=1 TO52:DK(I,1)=0:NEXT I
3010 FORI=1TO4:FORJ=1TO13
3020 EP=INT(RND(1)*52)+1:REM SELECT ENTR
Y POINT
3030 IF DK(EP,1)=0 THEN 3050
3040 EP=EP+1:IF EP>52 THEN EP=1
3045 GOTO 3030
3050 DK(EP,1)=J:DK(EP,2)=I:NEXT J,I
3060 RETURN
```

THE CORRECT FORM OF ADDRESS

We begin investigating how to program the 68000 chip with a look at the range of operand addressing modes it uses. This is an important aspect of programming the chip, especially when the operands you are dealing with are some form of structured data type — such as records and arrays.

In the introductory instalment of this series (see page 1697) we discussed the 68000 addressing capability with reference to the instruction set. In particular, we mentioned that although there is a wide range of addressing modes that can easily reference bytes, words or long words, we have to be very careful when using the addressing modes with particular instructions.

Consider the following 'generalised instruction':

```
OPCODE  Source, Destination
```

This is a 'semi-formal' way of describing what one class of instructions does with the source and destination operands. The sequence will involve getting the source operand (wherever that is, and whatever computation has to be done in order to get it), then perform the computation given by the opcode on that operand and deliver the result to the destination operand (again performing any computation that is needed to address the operand). For example, the instruction MOVEA D3,A6 results in the contents of D3 (the source) being moved to A6 (the destination). The opcode, MOVEA, indicates that an address is to be moved.

This is an example of very simple addressing in which no computation is required in order to address the operands. At the other end of the scale we could find that one of the operands may be addressed by adding the contents of an address register to a displacement integer and the contents of an index register (as in the Z80 LD r,(IX+d) instruction). We will discuss this in more detail later; for now, it is sufficient that there may be a fair amount of arithmetic computation involved in addressing operands.

Returning to our generalised model of the range of instructions, it is also possible that only one operand will be required for the opcode, as in:

```
OPCODE Source
```

For example, the branch instruction — as in BRA BACKHERE — requires only one operand (that is, the address to branch to BACKHERE). Finally, of course, we can have just:

```
OPCODE
```

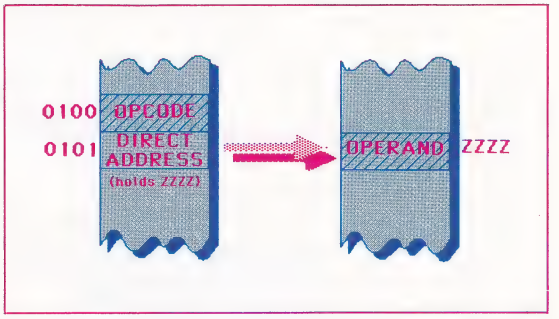
with no operands at all. A classic example of this form is the NOP instruction, which indicates that no operation is to be performed. In effect, this acts as a dummy instruction, and can be useful for hand patching or manually changing the program in memory.

Using this generalised 'model' of the range of instructions, the important thing to note is that where operands are concerned there may be some address computation involved. It is this computation that has to be specified by the programmer from the set of all computations or addressing modes available from the 68000.

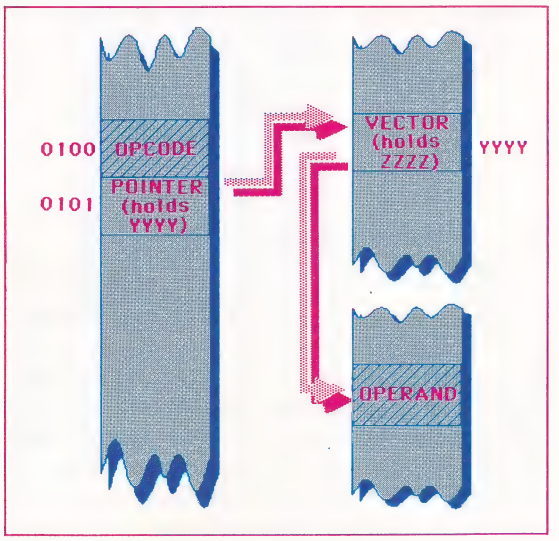
GENERAL ADDRESSING MODES

Most computers have at least five 'addressing modes' — the different ways by which operands can be addressed. Our diagrams illustrate the difference in operation of two of these modes:

- *Direct (or absolute) addressing:* In this mode, the memory address of the operand is stored with the instruction itself.
- *Indirect (or pointer) addressing:* The memory address where the address of the operand is stored is given with the instruction.



Direct Addressing
In this addressing mode, the opcode for the instruction is immediately followed by the address of a location in memory containing the operand data



Indirect Addressing
Indirect addressing requires the operation specified by the opcode to access its operand via a 'vector' address. The address of the vector immediately follows the opcode, and the vector holds the address of the required operand. This mode is particularly useful where the address of an operand is calculated during run-time, since only the contents of the vector need to be recalculated

As the diagrams show, in direct addressing the opcode will operate on an operand at address XXXX, while for indirect addressing, the operand will be found at location ZZZZ. The *pointer* in this case is held at memory location YYYY.

The other three major addressing modes are:

- *Immediate mode:* This is where a constant is held with the instruction. For example, in the instruction MOVEQ #25,D3 the constant 25 is addressed in the immediate mode.
- *Register mode:* In this, the operand is one of the register set and is specified with the instruction code itself. The operands in the instruction MOVE D2,D4 are the data registers D2 and D4.
- *Implied mode:* Here, the operands are implied by the instruction itself. So, in the case of RTS (ReTurn from Subroutine), the stack pointer and program counter are the implied operands.

68000 ADDRESSING MODES

Let's look closely now at the way the 68000 addresses it's operands. In addition to the general addressing modes already discussed, the 68000 has a 'program counter relative mode' (also known as 'PC relative'). Let's look at these modes in turn:

- *Absolute addressing:* We can access any memory location using this mode. The address of the operand appears after the instruction. For example:

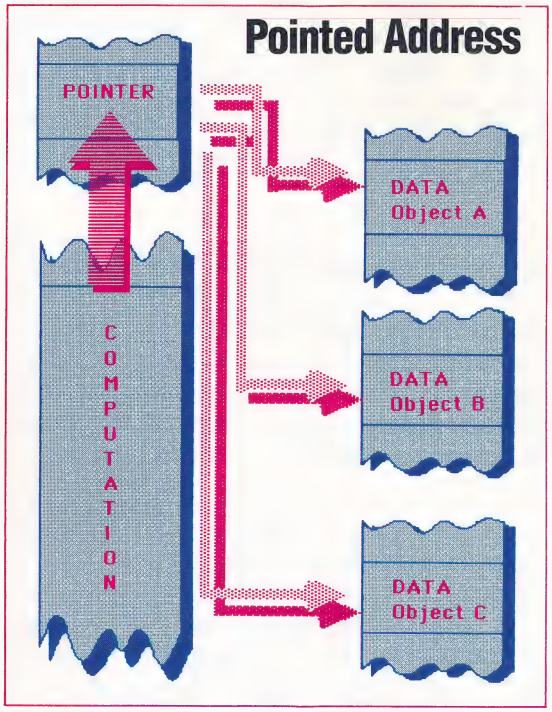
```
Address: Code:  Label:  Instruction:
1000     D678           ADD     DATA,D3
1002     2000
.
.
.
2000     0001   DATA    DC.W    1
```

It can be seen from this example that the symbolic name DATA has been given to address $2000, the instruction ADD absolute source (DATA) to D3 destination is coded as D678, and the absolute address DATA is contained at location $1002 (called the 'word extension'). Another example where there is only one operand is:

```
Address: Code:  Label:  Instruction:
1000     4278           CLR     COUNT
1002     3000
.
.
.
3000     0      COUNT   DS      1
```

Here the contents of location $3000 (COUNT) are cleared (set to zero) when the CLR instruction is executed.

In the previous instalment we mentioned that the PC was a 32-bit register (although only 24 bits are meaningful). This means that the absolute address that specifies the operand could be more than the one word given in the two examples above. How then does the assembler know how much space to allocate to the absolute address? It would obviously be wasteful to have a full long word extension for each absolute address reference, so what happens is that where the operand address is known (in the case of a backwards reference) the appropriate extension is used. Otherwise we have to specify to the assembler to use long or short word extensions.



Let's go back to the ADD example to show the effects of a long word extension:

```
Address: Code:  Label:  Instruction:
1000     D679   ADD     HIDATA,D3
1002     0020
1004     0000
```

In this example the absolute address of HIDATA is $200000. Notice that the code for the ADD part of the instruction remains the same (D6), but that the operand address part has changed from $78 to $79. The means by which we achieve this long word extension for absolute addressing will be dealt with in later articles on the assembler.

- *Register addressing:* This is the simplest form of addressing on the 68000; in this case the oprands are one of the set of registers. For example, ADD D0,D3, where the word in D0 is added to the contents of D3.

There are a few limitations in using this mode. It is not possible, for example, to have an address register as a destination for the ADD instruction — thus, ADD D0,A4 is not allowed. We can get round this, however, by using a different instruction, ADDA D0,A4, where ADDA is the add address instruction.

If we wish to use long words as our data objects in the above examples, we should include the attribute .L with the instruction: ADD.L D0,D3 would use the full 32-bit words as the data objects.

- *Register indirect addressing:* This mode is probably the most important mode on the 68000,

because it provides the 'pointer' mentioned earlier and also the means by which stack operations are performed. Let's consider the uses of the pointer first.

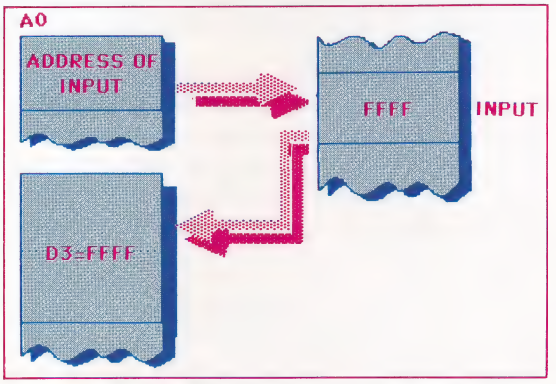
In the course of programming, we often need to point to a data object — either a byte, word, long word or a structured data object like a record or an array. We may then wish to repeat the computation or operation on another member of the same type of data object, and this is where the pointer is so useful. The pointer diagram shows how it can be used to reference different elements of a record. Initially the pointer directs the computation to be performed on the data object A; the pointer can then be reset to direct the computation to be performed on any of the other data objects.

The 68000 code to achieve this is via an address register pointer, rather than having the pointer stored next to the instruction, as with our general addressing modes example. This may be slightly inconvenient in some situations, but there are eight address registers available.

Now let's look at a simple indirect addressing mode example:

```
LEA      INPUT,A0
MOVE.W   (A0),D3
```

The LEA instruction loads A0 with the address of an input data object called INPUT, which is then copied into D3, as shown in the following diagram:



Now let's see how this mode is extended to operate on lists of data. First of all, we have the *post-increment extension.* Here, after the data object has been accessed by the pointer, it is incremented to point to the next item on the list. Let's examine the use of post-increment extension in an example where the data objects are words:

```
MOVE.W   (A0)+,D3
```

Here the pointer in A0 is incremented by two after the word pointed to by A0 has been accessed and copied to D3. So if we are pointing at address $2000 initially, then after the MOVE.W operation A0 will contain $2002. Of course, if we had been using byte objects then a MOVE.B operation would have only incremented A0 by one; and for a MOVE.L operation by four.

The power of this addressing mode is obvious if it is used in a program loop to perform some computation on each item in, say, a list. For example:

```
LEA       INPUT,A0     Initialise
MOVE.W    (A0),D3      Copy each item
            :
            :
          Computation
            :
            :
          Check end of list
```

Each time the computation is performed the next item in the list is automatically pointed to after each execution of the MOVE instruction.

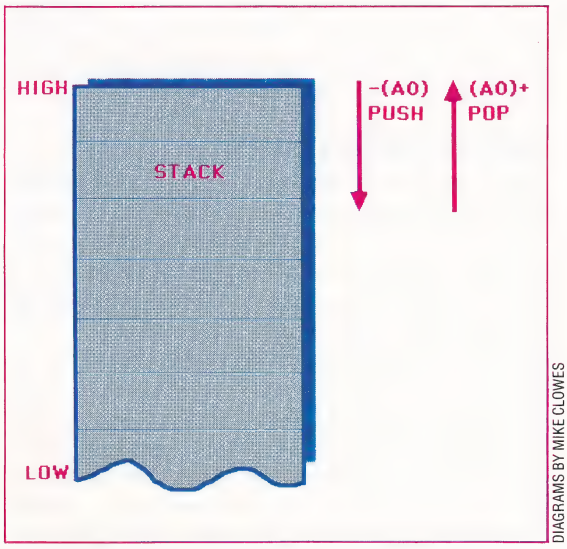
The other extension is called *pre-decrement.* Here, the address pointer is decremented before the data object is accessed. For example:

```
MOVE.W -(A0), D3
```

In this case A0 would be decremented by two before being used to copy the data object into D3.

Pre-decrement indirect addressing is the complementary addressing mode to post-increment indirect. Post-increment progresses through a list in the order of increasing addresses, while pre-decrement works in reverse order. An important point to note here is that we cannot separate the pre- and post-operations for our own convenience. For example neither (A0)- nor +(A0) is allowed. Instead, we must stick to the specified pre-decrement -(A0) and post-increment (A0)+.

You may have noticed how the pre-decrement and post-increment addressing modes are in a sense stack operations. We use these operations to 'push on' to a stack and 'pop off' a stack where a stack is, as conventionally defined, a series of high to low addresses. In this way, the MOVE D0,-(A0) instruction is a 'push on' and a MOVE (A0)+,D0 is a 'pop off' operation — as illustrated in our diagram.



DIAGRAMS BY MIKE CLOWES

Stacked Address
The PUSH and POP instructions familiar to Z80 programmers can be simulated using the powerful pre-decrement and post-increment extensions of the 68000, which can be used to perform a PUSH and POP, respectively

We will be looking at stacks and how they are used on the 68000 later in the series. For the moment, it's enough to note that we have a very convenient addressing mode to access lists of data, be they stacks or more structured data.

WHAT'S ON THE MENU?

We continue our series on word processing with a look at MicroPro's WordStar system, which although released in the late 1970s is still the industry leader. Based largely on the CP/M operating system, WordStar's enormous range of capabilities makes it an extremely powerful tool.

Literally hundreds of word processing programs have been written over the last decade, and each one has claimed to be an improvement over the rest. Yet the market leader remains one of the oldest word processors around. WordStar, from MicroPro, was originally written for CP/M machines, and made its first appearance in 1978, shortly after the company was founded. It has since been rewritten to run under PC-DOS/MS-DOS, and was chosen for the IBM PC and, consequently, for its compatibles, which now dominate the business market. WordStar's staying power is due to its almost unrivalled word processing facilities. This is in spite of the fact that the system is not particularly user-friendly, although it did pioneer the use of on-screen 'help' menus.

As we've just noted, the roots of WordStar lie in the CP/M operating system (for more information on CP/M, see our series beginning on page 1264), and as such, it carries many of that system's best and worst features. To begin with, when WordStar is loaded, it is added to the list of CP/M's 'transient' programs, which lets WordStar take advantage of CP/M's disk operating facilities. And like CP/M, WordStar makes use of a wide range of control characters.

THE OPENING MENU

After loading and running the program, WordStar users are confronted with the Opening Menu, which displays all the options available, as well as the contents of the currently logged disk drive. If you haven't changed the disk, this will consist of the programs that make up WordStar. It's worthwhile noting here that WordStar filenames use the same format as CP/M filenames — a name of eight characters or less followed by a full stop and a three-letter extension if required.

On the Opening Menu is a list of 13 commands, which are divided into five sections. Under Preliminary Commands, you have the choice of changing the logged drive, turning the File Directory on and off, and setting the Help level. The Help level is defaulted to level 3 on booting up, and displays all the available commands at the top of the screen. By moving down to level 0, the screen will be clear except for the status line.

You have the choice at this stage of opening or editing a document or non-document file. The former is text that can be edited with WordStar while the latter refers to programs that can be run from your computer. A list of File Commands follows including the options to print, rename, copy and delete.

The last two sections in the Opening Menu are System Commands and WordStar Options. The first provide you with an interface to CP/M, allowing you to run one of the programs on disk, or else exiting to the operating system. The second give you the choice of using either MailMerge or SpellStar. MailMerge is a mailing list program ideal for printing out personalised letters, addresses for envolopes, and other business-orientated mailing activities. SpellStar is a spelling checker that checks each input word in a document file against a dictionary held in memory. Any words that don't correspond with those in the dictionary will be pointed out to the user for confirmation.

Using the D option at this point will result in a document file being opened (or one being created if it is a new file), and the screen will switch back to the Main menu. Indicated on the top line is the currently logged drive, and the name of the document file to be edited. This is followed by the current cursor position and a prompt informing you whether or not the Insert facilitiy is ON or OFF.

Essential Abilities

Listed here are 11 features available on many word processing programs. In this and the following instalments, we'll be looking at whether the packages examined contain these features and how well they are implemented.

Wordwrap
The ability of a program to move a word from one line to the next if there is insufficient space.

Block Movement
Allows the user to define a 'block' of text that can be manipulated independently of the rest of the document.

On-Screen Help
Provides assistance, in the form of messages, informing the user how to access the features available.

80-Column Screen
A word processing package should allow the user as much view of the document as possible. An 80-column screen is considered the minimum for this.

Word Count
This gives the user an indication of how much has been written so far.

Find/Replace
Searches for letters, words or phrases and changes them to something else.

WYSIWYG
This is an acronym for 'what you see is what you get' and refers to the process whereby the user can view text on the VDU in the form in which it will appear on the printed page.

Mailshot Facility
Installed either integrally or as an associated program, mailshot programs can 'customise' standard letters or documents by inserting specific names and addresses at the required points and printing the address labels.

Spelling Checker
Provided in a similar format to mailshot programs, spelling checkers will read each word in a document and compare it with a dictionary of words held in memory. Words that do not match with those in the dictionary will be pointed out to the user.

Founts Available
Although a lot will depend on the printer, many word processors will support different founts, such as bold and italic.

File Linking
Documents are necessarily limited by the amount of available memory. Thus, when producing long documents, it is useful to be able to link the files together for printing or Search/Replace functions.

A Galaxy Of Stars

As WordStar has become so popular, it is not surprising to discover that its manufacturer, MicroPro, has released several versions of the original CP/M program. Over the past few years, one of the most popular versions of WordStar has been WordStar Professional, which has been translated to run under the MS-DOS system used on the Apricot range and the IBM PC. As an added incentive, MicroPro has included SpellStar and MailMerge.

More recently, the company has launched WordStar 2000, also for the IBM PC. This program, although superficially similar to its ancestor has been described as 'the Rolls Royce of word processors'. Provided on five disks, the programs and files that constitute WordStar 2000 add up to two Mbytes! But this kind of power is not cheap — WordStar 2000 is priced at £506.

Finally, as an indication of the future of the CP/M WordStar, MicroPro has written a version of the program known as Pocket WordStar intended for users of the Amstrad CPC 464 and 664. Because these computers maintain a high-resolution screen display, there is insufficient memory remaining from the 64 Kbytes available to be able to run WordStar adequately. Therefore, MicroPro has developed a smaller version for these machines, which will be marketed by Cumana

Below this, assuming that the Help level is still set at 3, is another list of available commands, once again divided into sections. The first displays a summary of the cursor movements, amply illustrating WordStar's attachments to CP/M. Early CP/M machines didn't have cursor control keys, and so movements were performed by pressing the Control key plus another key simultaneously. These keys are concentrated on the left-hand side of the keyboard, the core being the S, E, D and X keys, which correspond to cursor left, up, right and down, respectively.

Among the miscellaneous items, you will notice B (CTRL B), which is listed as Reform. This will rejustify a paragraph on screen after editing and corrections have made a mess of the original. (Incidentally, the format appearing on screen corresponds exactly to how it will appear on hard copy, providing, of course, that your printer accommodates whatever margins have been set in WordStar.)

On the far right of the Main menu help screen are a series of other menu titles, giving you access to a number of additional functions. The Help menu, for example, provides detailed descriptions of how each WordStar command works. The Quick menu, on the other hand, contains a number of miscellaneous 'quick' commands, including, among others, the various Find and Replace options.

The On-screen menu provides the means to format the screen to any size and shape, letting you set the tabs, margins, line spacing and so on. One very useful function allows you to set the page length, appearing as a dotted line in the text. With this, you can see where the page breaks will occur when printing and so enabling you to arrange the document accordingly.

Also available on the On-screen menu is the Wordwrap toggle. Once set, this takes any word that will not fit onto the end of a line and places it at the beginning of the next, thereby removing the worry and bother of hyphenating words.

The third of the additional menus contains some of the most useful and powerful commands available on WordStar. Although the Block menu can perform a number of operations, its principal function is to manipulate blocks of text within a document. The range of operations is listed under the Block Operations section.

By performing CTRL commands at the beginning and end of required text, blocks can be created and will be displayed in 'reverse video' (the blocked text will be significantly less bright than the rest for easy recognition). Once the block has been created, it becomes possible to move or copy the text to another area of the document, delete it or even save it to a separate file on disk. But you're not confined to moving only paragraphs. It's also possible under WordStar to move columns, which is very handy if you are writing tables or letters with two or more rows of text. In this case, it is merely a question of demarcating for blocks columns instead of lines.

Included in the Block menu are the three save

Wordwrap

★★★★

Although WordStar wordwraps when text is being written, it does not automatically reformat text when margins are changed.

Block Movement

★★★★★

WordStar allows the definition of blocks of any size and permits the full range of block manipulations.

On-Screen Help

★★★★★

On-screen Help menus are characteristic of WordStar and helped establish its popularity.

80-Column Screen

★★★★★

The program not only supports an 80-column screen, but also permits margins of up to 255 characters across.

Word Count

There is no word-counting facility in WordStar.

Find/Replace

★★★

WordStar supports several versions of this feature, enabling strings of up to 30 characters to be searched for.

WYSIWYG

★★★★★

More than any other word processing package, WordStar is acclaimed for its ability to format text on screen.

Mailshot Facility

★★★★

WordStar was one of the first programs to feature a mailshot program, and MailMerge is still the standard by which others are judged.

Spelling Checker

★★★★

Advanced versions of SpellStar have a dictionary of around 20,000 words to use for spelling verification.

Founts Available

★★★

Bold Strike and italics are included in the additional founts.

File Linking

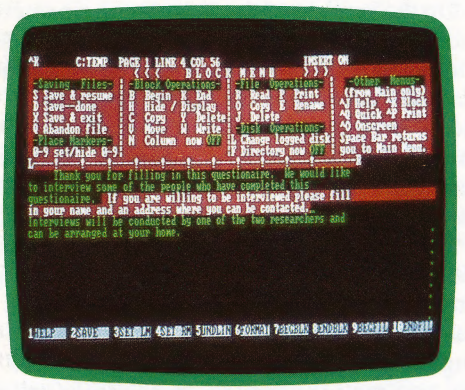
★★

Separate files can be printed continuously using MailMerge, but facilities such as Find/Replace through several files at once are not supported.

À La Carte

WordStar is characterised by the Help menus, which are displayed at the top of the screen. Although experienced users tend to dispense with the Help menus, they are extremely useful to beginners, as they provide an easy reference to the commands that are available.

Each of the three screens shown here provides a different range of commands. The opening screen, which is the one presented to the user when WordStar is loaded, contains mostly DOS commands to enable the user to access files. The Main menu contains commands that are most likely to be used when keying in the text, while the Block menu contains a number of editing commands



This prompt tells the user the current logged drive and the file that is being edited

This is the Main menu under Help level three, giving full explanations of the available commands

Page, line and column numbers indicate the cursor position

This prompt reminds you that INSERT mode is ON

IBM versions of WordStar allow a number of commands to be entered via the function keys. This line shows the current function key settings

This line shows the number of columns currently set, although these can be altered. The ! signs indicate the default TAB settings, which can also be changed

commands, offering the options of returning to the text, going to the main menu or exiting WordStar. You should bear in mind that when using commands listed under the separate menus, it's not necessary to recall the menu to the screen. To save and exit the system, for instance, you merely have to type CTRL KX, which will perform the command automatically.

Let's conclude this brief overview of WordStar with a look at the 'dot commands'. These are embedded in the text on separate lines, each beginning, naturally enough, with a full stop, and concern print formatting and adding extra features to the hard copy, such as top and bottom margin widths and page numbers. The command .pl (followed by a number), for example, sets the number of lines to be printed on a page. Because these commands are not acted upon by WordStar until the software is processing the document for printing, they in fact act as CTRL characters to the printer.

WordStar's power has managed to keep this program at the top of it's field for almost a decade. As CP/M moves 'down market' towards home machines such as the Amstrad range, it seems very likely that WordStar will follow and become the leader in the home computer market as well.

A Popular Bundle

WordStar is an interesting example of how the recognised popularity of a program generates its own momentum to make it even more popular. Once WordStar established itself as the leading word processing software for CP/M micros, many manufacturers began bundling it with their machine, resulting in an even larger user base for the package.

An early example of this kind of marketing was the Osborne 1, which not only bundled WordStar with the machine, but also included the SuperCalc spreadsheet. The fact that both of these programs run under CP/M meant that it was possible for them to share common files. This kind of idea proved so popular that other software programmers began to incorporate the idea of data passing and file sharing, which eventually led to the development of integrated software packages, such as Lotus 1-2-3.

Nowadays, almost all micros come with bundled software, particularly word processors. These range from the highly sophisticated and very expensive Perfect Software from Thorn EMI (for the Advance and Wren) to the budget-priced Tasword II, which was bundled in the 'six-pack' provided with early versions of the Spectrum+

Motorola 68000 Family Fact Sheet

Here, courtesy of Motorola Inc, we present information about the 68000 family of semiconductor devices

	DEVICE NUMBER	DEVICE NAME	DESCRIPTION	SPEEDS	PACKAGE TYPE
PROCESSORS	MC68000	16/32 bit MPU	16 bit external/32 bit internal MPU. 17 general purpose 32 bit registers. 16 MB linear address space.	8, 10, 12.5 MHz	64 lead L, LC, P 68 lead R FN (4Q85)
	MC68008	8/32 bit MPU	8 bit external/32 bit internal MPU. 17 general purpose 32 bit registers. 1 megabyte linear address space (4 MB with FN package option).	8, 10 MHz	48 lead L, P 52 lead FN (1Q86)
	MC68010	Virtual Machine 16/32 bit MPU	16 bit external/32 bit internal MPU. 17 general purpose 32 bit registers. Virtual memory/machine. 16 megabyte linear address space.	8, 10, 12.5 MHz	64 lead L, LC, 68 lead R, RC
	MC68012	Extended Virtual 16/32 bit MPU	16 bit external/32 bit internal MPU. 17 general purpose 32 bit registers. Virtual memory/machine. 2 gigabyte linear address space.	8, 10, 12.5 MHz	84 lead RC
	MC68020	32 bit MPU	Complete 32 bit microprocessor. 4 gigabyte linear address space. Coprocessor Interface. Instruction Cache. Dynamic Bus Sizing. 2-3 MIPS performance.	12.5, 16.67 MHz	114 lead RC
	MC68881	Floating Point Co-processor (FPCP)	Meets full IEEE spec for advanced floating point calculations. Single, double and extended precision.	12.5, 16.67 MHz	68 lead RC
MEMORY MGMT	MC68451	Memory Management Unit (MMU)	Ideal MMU for non-demand paged MC68000, 68010 systems.	8, 10 MHz	64 lead L, LC 68 lead R, RC
	MC68851	Paged MMU (PMMU)	32 bit demand virtual paged memory management unit for MC68020 based systems. Available 1st half '86.	12.5, 16.67 MHz	124 lead RC
DMA CONTROL	MC68440	Dual DMA (DDMA)	Dual channel, high speed Direct Memory Access controller. Capable of 5 MB/sec data transfer rates.	8, 10, 12.5 (3Q85) MHz	64 lead L, LC, P 68 lead R, RC
	MC68450	DMA Controller (DMAC)	Four channel Direct Memory Access Controller. Capable of very complex 'chained' data tranfers.	8, 10 MHz	64 lead L, LC 68 lead R, RC
	MC68442	Expanded DDMA	32 bit address version of DDMA. Support for 4 gigabyte range of MC68020. Pin compatible with 68440/68450 (4Q85).	8, 10, 12.5 MHz	68 lead R
SYSTEM INTERFACE	MC68153	Bus Interrupt Module (BIM)	Routes interrupts from 4 independent sources to any of 7 M68000 MPU interrupt levels. Bipolar.	200 ns access time (16 MHz clock)	40 lead L, P
	MC68452	Bus Arbitration Module (BAM)	Arbitrates access of an M68000 system bus between up to 8 local masters. Bipolar.	50 ns arbitration time	28 lead L, P
DATA COMMUNICATIONS	MC68652 *MC2652*	Multi-Protocol Comm Controller (MPCC)	Single channel. Byte control. Bit oriented. CRC (error correction) circuitry. (MC2652 has generic bus interface.)	2 Mb/s	40 lead L, P
	MC68653 *MC2653*	Polynomial Generator Checker (PGC)	Error correction, code generation/comparitor circuit. Excellent companion chip for 68652. MPCC or 68661 EPCI. (MC2653 has generic bus interface.)	4 Mb/s	16 lead L, P
	MC68661 *MC2661*	Enhanced Peripheral Comm I/F (EPCI)	Universal synch/asynch. Double buffered receiver/transmitter. Internal baud rate clock. (2661 generic bus I/F)	1 Mb/s	28 lead L, P
	MC68681 *MC2681 MC2682*	Dual UART (DUART)	Dual channel. Quad buffered receiver. Double buffered transmitter. Independent baud rate selection. 1 Mb/s. (2681 has generic bus interface; 2682 offers partial functionality in a smaller package.)	1 Mb/s	40 lead L, P (MC2682-28 lead)
GENERAL I/O	MC68120/ 68121	Intelligent Peripheral Controller (IPC)	Provides peripheral control for MC68000 or M6800 systems.	1, 1.25 MHz	48 lead L
	MC68230	Parallel Interface/ Timer (PI/T)	Unidirectional/bidirectional 8/16 bit, double buffered parallel interface. 24 bit timer with 5 bit prescaler. M68000 I/F.	8, 10 MHz	48 lead L, P
	MC68901	Multi Function Peripheral (MFP)	Single channel USART. 8 source interrupt controller. 8 parallel I/O lines. Four 8 bit timers. Introduction 3Q85.	4 MHz 1 Mb/s USART	48 lead L, P
GRAPHICS	MC68486 (RMI) MC68487 (RMC)	Raster Memory System (RMS)	Provides functionality required by bit mapped or object-oriented graphics systems. Features include object definition & manipulation, collision detection, light pen input, x/y capture & interrupt, 32 of 4096 colours, visual/virtual screens. Introduction 1st half '86.	N/A	48 lead L, P (both)

Key To Package Types: L = Ceramic DIP, LC = Ceramic DIP, Gold Lead Finish; P = Plastic DIP; R = Pin Grid Array, Gold Lead Finish; ZB = Socketable LCC; FN = Plastic Quad Pack

© 1983 GORDON, D.—RANDALL-GORDON ASSOC.